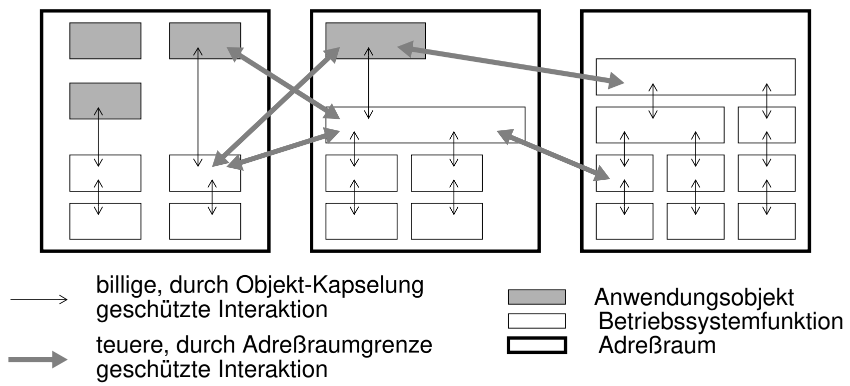
G.4 Objektbasierte, offene Betriebssysteme (2)

+ Strukturierungsmöglichkeiten von Systemen mit ausgelagerten Komponenten mit Effizienz eines monolithischen Kerns kombinierbar
  - Da die Modularisierung nicht mehr durch Adressraumgrenzen, sondern durch programmiersprachliche Methoden erzwungen wird, kommt es zur Laufzeit gegenüber einem monolithischen Kern zu wenig Effizienzeinbußen.

+ Individuelle Anpassbarkeit wie bei einem Minimalkern möglich
  → durch feinere Granularität der Module jedoch weit flexibler
  - In einem Minimalkernsystem können lediglich komplette, ausgelagerte Module (z. B. ein weiteres Dateisystem) an Anwendungsanforderungen angepasst werden und dann neben dem "Standard-Modul" installiert werden.
  - Der objektbasierte Ansatz erlaubt eine feinere Modularisierung.
    - es können damit auch feinere Module anwendungsspezifisch angepasst und installiert werden
    - unveränderte Komponenten des Systems können von den angepassten Komponenten mitverwendet werden
  - Wenn auf Sprachebene zugesichert werden kann, dass ein Modul nicht unkontrolliert auf seine Umgebung zugreift, so kann solch ein Modul gefahrlos auch in einen privilegierten Adressraum installiert werden (in dem es selbst zwar keine privilegierten Aktionen ausführt, aber effizient mit Systemkomponenten interagieren kann, die ihrerseits dann privilegierte Befehle benutzen).
    - die Funktion solcher Module wird dann nur von der Anwendung genutzt, die sie installiert hat, der Rest des Systems wird nicht tangiert
    - das System ist damit anwendungsspezifisch erweiterbar — es wird zum offenen System



G.5 Anforderungen an zukünftige Betriebssysteme

- Betriebssysteme müssen an die Anforderungen von Anwendungen anpaßbar sein
  - *General purpose* Betriebssysteme realisieren häufig "Kompromisse" für Betriebssystemfunktionalität, die für bestimmte Anwendungen ungeeignet sind
    - Effizienz
      - Für nummerische Anwendungen ist es beispielsweise wichtig, einen möglichst großen Anteil der Prozessorleistung zu erhalten — Schedulingstrategien, die ein gutes interaktives Verhalten des Rechners unterstützen oder auch gute Schutzmechanismen sind dabei hinderlich.
    - Schutzanforderungen
      - Bei der Verarbeitung sicherheitskritischer Datenbestände sind Effizienzüberlegungen gegenüber guten Schutzmechanismen zurückzustellen.
    - Zugriffs- oder Cache-Strategien
      - Aus Effizienzgründen implementieren Betriebssysteme Cachemechanismen (z. B. für Zugriffe auf Sekundärspeicher), die für den Anwender nicht sichtbar werden. Es gibt aber Anwendungen, die in bestimmten Situationen sicher sein müssen, daß Daten tatsächlich auf den Hintergrundspeicher geschrieben wurden (z. B. Datenbanken im Rahmen des Commit eines Transaktionsprotokolls).

  ➜ Anwendungen dürfen nicht gezwungen sein, Betriebssystemmechanismen zu umgehen
  - Es ist häufig zu beobachten, dass Anwendungen durch Tricks versuchen, Mechanismen des Betriebssystems zu umgehen — das kann nicht der Sinn eines Betriebssystems sein!

- Höheres Abstraktionsniveau bei der Programmierung von Betriebssystemkomponenten
  - Wenn auch bei der Programmierung von Betriebssystemkomponenten auf mächtige Abstraktionen aufgebaut werden kann (z. B. einheitliche Modulinteraktion lokal und Adressraum-übergreifend oder leichtgewichtige Prozesse), kann die Programmierung auf einem höheren Abstraktionsniveau erfolgen.

  ➜ Anpassung, Erweiterung und Wartung von Betriebssystemen einfacher

- Betriebssysteme für verteilte Systeme, Mehrprozessorsysteme, Parallelrechner
  - Betriebssysteme müssen für moderne Hardwarearchitekturen geeignet sein und dem Anwender deren Eigenschaften in Form von gut handhabbaren Abstraktionen zur Verfügung stellen.

G.6 Objektorientierte BS: Choices

CaI93. Roy H. Campbell and Nayeem Islam, "CHOICES: A Parallel Object Orientd Operating System", in Gul Agha, Peter Wegner, and Akinori Yonezawa [Eds.]: *Research Directions in Concurrent Object-Oriented Programming,* MIT Press, Cambridge, Massachusetts, 1993.

CIRM93. Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany, "Designing and Implementing CHOICES: An Object-Oriented System in C++", *Communications of the ACM*, Vol. 36, No. 9, pp. 117 - 126, Sep. 1993.

- Grundidee: Objektorientiertes Betriebssystem-Framework
  - entwickelt seit 1987 an der University of Illinois at Urbana-Champaign
  - Implementierung in C++
    - Sun SPARCstation I + II, Encore Multimax, Emulation auf UNIX
    - über 300 Klassen, ca. 150 000 Zeilen Code
  - beschreibt Frameworks für
    - Prozessverwaltung
    - virtuelle Speicherverwaltung
    - Dateisysteme
    - Netzwerk

1 Beispiel: Choices-Prozessverwaltung

- Komponenten
  - MemoryObject
    - Datenspeicher
  - Process
    - Aktivitätsträger zur sequentiellen Ausführung eines Algorithmus
  - Domain
    - Umgebung, die Namen, die von einem Aktivitätsträger zur Benennung von Daten benutzt werden (z. B. virtuelle Adressen) in Speicheradressen abbildet
      (ein Domain ist damit eine Abstraktion eines virtuellen Adreßraums)
- Beziehungen

- Einem Domain sind mehrere Prozesse zugeordnet (➔ leichtgewichtige Prozesse oder Threads). Es können dynamisch weitere Prozesse in einem Domain hinzukommen — d. h. es können neue erzeugt werden.
- Ein Prozeß ist immer genau einem Domain zugeordnet. Die Zuordnung ist nicht dynamisch veränderbar — d. h. Prozesse können nicht zwischen Domains wandern.
- Jedem Domain sind mehrere Memory Objects zugeordnet (das entspricht den Segmenten).
- Ein Memory Object kann mehreren Domains zugeordnet sein (= Shared memory).
- Die Zuordnungen können dynamisch verändert werden. Es können somit neue Memory Objects in einem Domain hinzukommen (= Speicher nachallokieren), Memory Objects aus einem Domain entfernt werden (= Speicher freigeben oder Shared memory deallokieren) und Memory Objects in weiterer Domains abgebildet werden (Shared memory Zuordnungen verändern).

2 Architektur

- konventionelle Systemarchitektur
  - ➤ privilegierter Systemkern
  - ➤ Anwendungen in Benutzermodus
- einfache Erweiterbarkeit durch objektorientierten Entwurf und Framework-Konzept
  - ➤ Realisierung kompatibler Subklassen von existierenden Systemklassen
- einheitliches Schnittstellenkonzept
  - ➤ innerhalb von Anwendungen
  - ➤ zwischen Anwendung und Betriebssystem
  - ➤ zwischen Anwendungen

3 Systemschnittstelle

- Ziel: Systemaufrufe = Methodenaufrufe an Objekte des Betriebssystems
- Probleme:
  - ➤ Aufrufer im Benutzer-Adreßraum, Ziel im Kernel
  - ➤ Referenz (Zeiger) auf BS-Objekt
  - ➤ Parameterübergabe
- Konventionelle Lösung: spezielle Funktionen & SVC/trap-Befehl
- Lösung in Choices:
  - ➤ Proxy-Objekte: Stellvertreter für Systemobjekte
  - ➤ Nameserver: Systemaufruf-Name → Proxy-Referenz
    - Anwendungsobjekte erhalten Referenzen auf Proxy-Objekte mit Hilfe der lookup-Methode des NameServers. Das Anwendungsobjekt benennt das gewünschte Systemobjekte dabei über einen symbolischen Namen.
    - Die NameServer-Schnittstelle wird durch eine abstrakte Klasse NameServer festgelegt - es sind verschiedene konkrete Unterklassen denkbar, die die Abbildung unterschiedlich vornehmen.
    - Jeder Anwendung wird initial ein *default*-NameServer zugeordnet.
    - Das Betriebssystem kann mit Hilfe der bind-Methode neue Systemobjekte bei einem Nameserver registrieren.
    - Nur die speziell als *proxiable* gekennzeichnete Schnittstellen registrierter Systemobjekte sind aus Anwendungen heraus aufrufbar.
    - Dem Benutzer werden *Header-Files* mit den *proxiable*-Schnittstellen der aufrufbaren Systemobjekten zur Verfügung gestellt.
    - Beim Aufruf von Methoden von Systemobjekten werden Sicherheitsprüfungen durchgeführt.

4 Proxy-Objekte

Proxy-Objekte als Systemaufruf-Schnittstelle

- abgebildet in *read-only*-Adreßbereich des Benutzeradressraums
  - ➔ Zustand des Proxy-Objekts vom Anwendungsprogramm nicht modifizierbar
  - ➔ Proxy-Objekte zwischen verschiedenen Domains *sharable*
- lösen über trap Einsprung in Systemkern aus
  - Anwendungszustand wird dabei gesichert
  - Methodennummer, Aufrufparameter und der *this*-Zeiger werden für Kern abgelegt
  - Systemkern ermittelt aufzurufendes Systemobjekt aus gesichertem Anwendungszustand
- Referenzen (Zeiger) auf Proxy-Objekte an Systemobjekte übergebbar

  Proxy-Objekte auch im Systemkern gültige Stellvertreter für Systemobjekte
  - von Parametern, die für Proxy-Objekte stehen, wird nach Einsprung in den Kern die Proxy-Information entfernt:
    - ➔ entweder bleibt eine Referenz auf ein Kern-Objekt

    oder das Zielobjekt liegt in einem Anwendungs-Domain
    - ➔ Kern erzeugt Proxy hierfür, so dass bei einem Aufruf in die entsprechende Domain geschaltet wird
- neue Systemobjekte instantiierbar
  - Falls die Konstruktor-Methode einer Systemklasse als *proxiable* deklariert ist, können Objekte dieser Klasse aus einer Anwendung heraus instantiiert werden. Das Objekt selbst entsteht dabei allerdings im Systemkern, in der Anwendung entsteht lediglich ein Proxy-Objekt. Bei einem delete wird das ProxyObjekt gelöscht, das dahinter stehende Systemobjekt nur dann, wenn es nicht mehr anderweitig referenziert wird.

G.7 Objektorientierte BS: JX

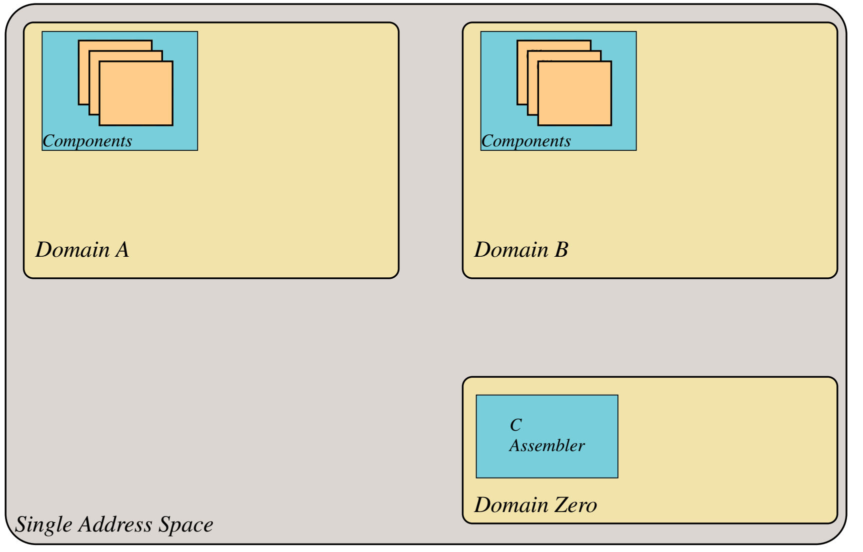
1 Grundkonzepte

- Java-basiert
- Single-Adress-Space Betriebssystem
  - ◆ Schutz durch Sprachmechanismen statt durch MMU
    - ➤ Typkonzept
    - ➤ erweiterter Byte-Code-Verifier
    - ➤ spezielle Compiler-Unterstützung
  - ➥ feingranulare, flexible und effiziente Einheiten
- Betriebssystem dynamisch erweiterbar
  - ◆ Kapselung auch zwischen Betriebssystemmodulen
  - ➥ Erweiterungen muss nicht blind vertraut werden

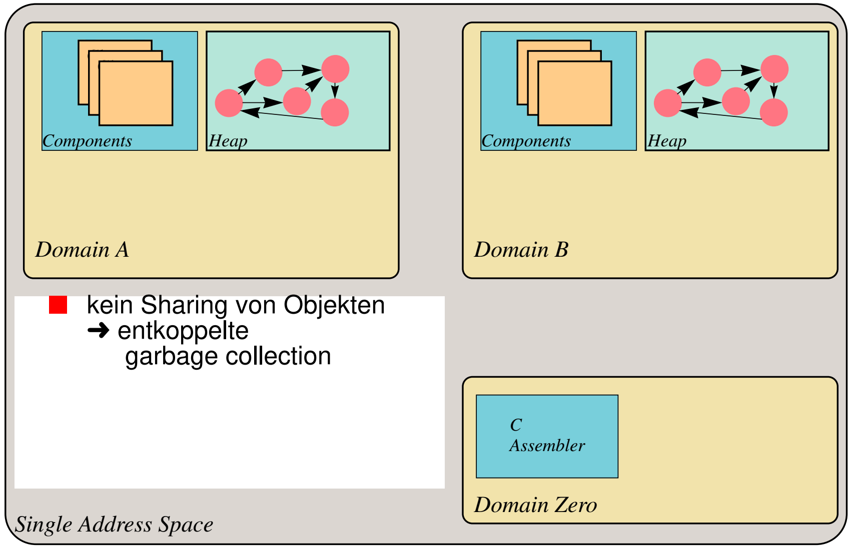
2 Architektur

- Domains
  - ➤ Schutzeinheit
  - ➤ Ressourcenverwaltung
  - ➤ Eingrenzung von Fehlverhalten
  - ➤ unabhängige Terminierung
- Komponenten
  - ➤ Code-Module (Java-Bytecode), werden in Domain geladen
  - ➤ Bytecode wird durch erweiterten Verifier überprüft
  - ➤ werden zur Ladezeit in Maschinencode übersetzt
- DomainZero
  - ➤ enthält in C und Assembler geschriebene Komponenten
  - ➤ kein Kern im klassischen Sinn — andere Domains setzen nicht darauf auf, sondern nutzen seine Dienste

2 Architektur (2) — Domains und Komponenten

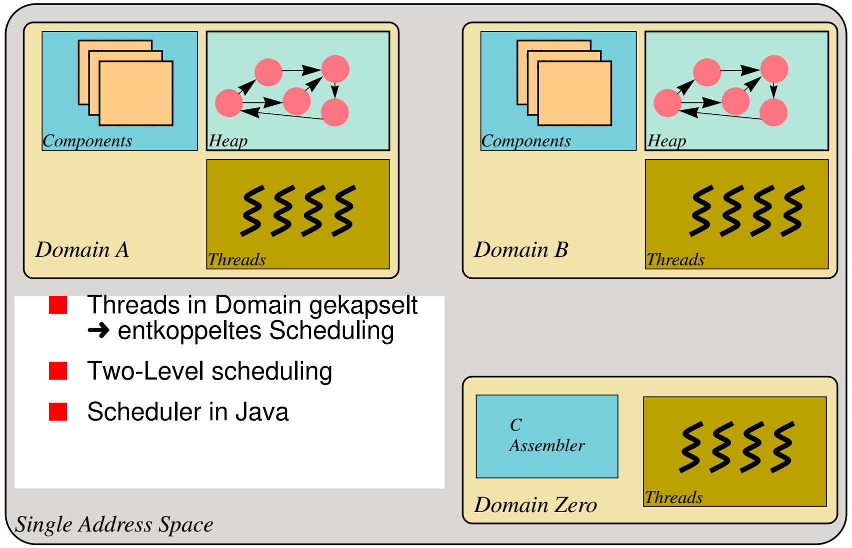


2 Architektur (3) — Objekte und Heap



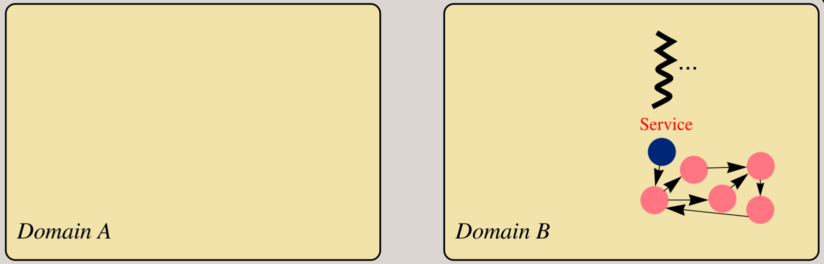
- kein Sharing von Objekten
  ➜ entkoppelte
  garbage collection

2 Architektur (4) — Threads



- Threads in Domain gekapselt
  ➜ entkoppeltes Scheduling
- Two-Level scheduling
- Scheduler in Java

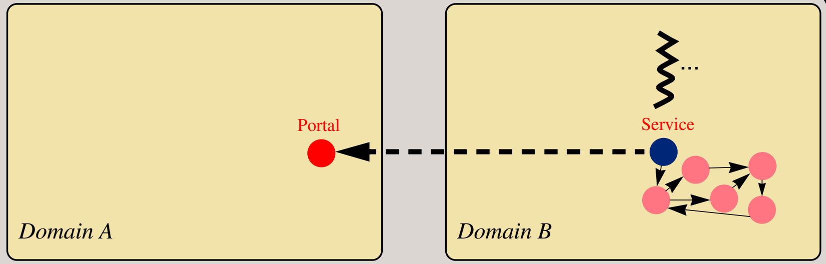
3 Inter-Domain-Kommunikation — Portale



- RMI-ähnliche Kommunikation
- Domain exportiert "Service"
- Schnittstelle wird durch Java-Interface beschrieben
- Service ist mit Service-Thread(s) verknüpft

Single Address Space

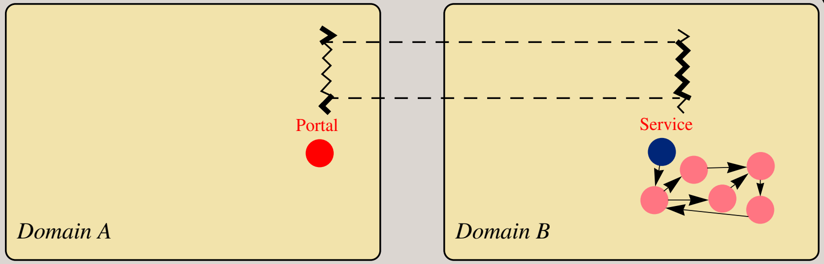
3 Inter-Domain-Kommunikation — Portale (2)



- Service wird bei Name-Service registriert
- Portal wird erzeugt

Single Address Space

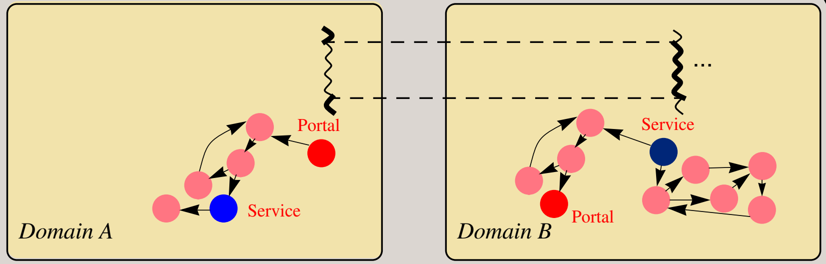
3 Inter-Domain-Kommunikation — Portale (3)



- Portal ist ein Proxy
- Bei Aufruf Thread-Umschaltung auf Service-Thread (Rendezvous, hand-off-Scheduling)
- Keine Thread-Migration

Single Address Space

3 Inter-Domain-Kommunikation — Portale (4)



- Parameterübergabe:
  - ➤ "normale" Objekte werden by-value übergeben
  - ➤ für Service-Objekte wird im Ziel-Domain ein neues Portal erzeugt

Single Address Space

3 Inter-Domain-Kommunikation — Portale (5)

- Kosten im Vergleich zu IPC-Mechanismen in anderen Forschungsbetriebssystemen:

System	IPC (cycles)
L4KA (PIII, sysenter, sysexit) (PIII, 450MHz)	800
Fiasco/L4 (PIII 450 MHz)	2610
Alta/KaffeOS	27270
JX/native (PIII 500MHz)	691

- Kosten in JX immer noch Faktor 20 gegenüber normalem Methodenaufruf
  - ➤ aber reiner Softwareschutz
  - ➤ weitere Optimierungen möglich

4 Optimierungen: Fast Portals

- Methodenaufrufe an Domain Zero
  - ◆ Domain Zero muss ohnehin vertraut werden (C- oder Assembler-Code)
    - ➤ kontrollierte Domain-Grenze überflüssig
    - ➤ einfacher Methodenaufruf ohne Threadwechsel
- Inlining
  - ◆ Inlining von kleinen Methoden durch den Compiler
- Spezielle Memory-Objekte

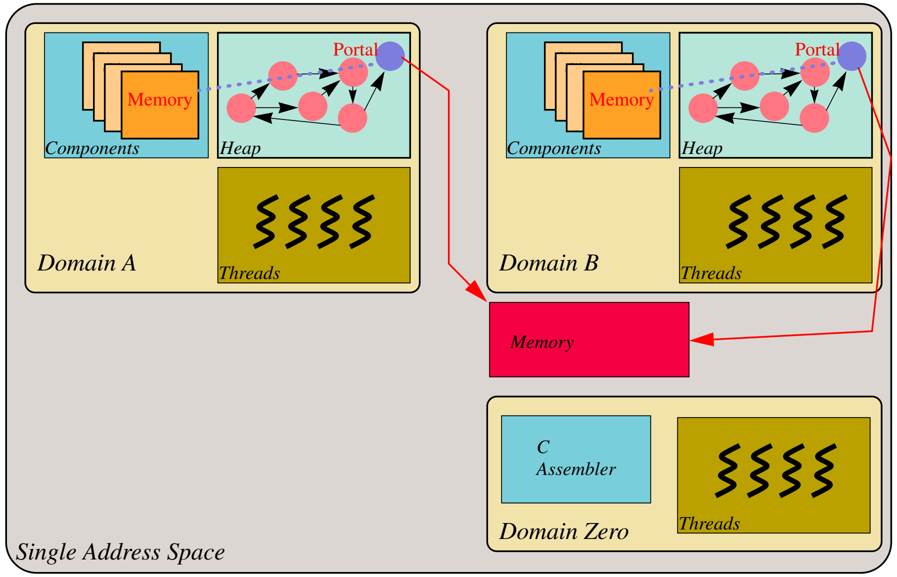
5 Optimierungen: Memory Objekte

- Probleme mit Arrays:
  - ➤ Unterbereiche nur als Kopie
  - ➤ keine Zugriffskontrolle möglich

➜ Memory-Objekte

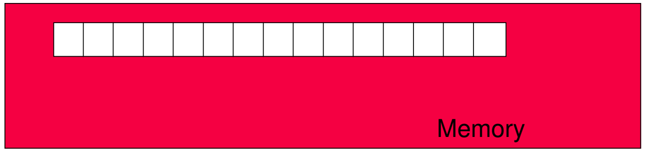
- ◆ Interface **`Memory`**
- ◆ Memory-Interface darf von keiner Klasse implementiert werden (Verifier)
- ◆ Memory-Aufrufe werden vom Compiler als direkter Speicherzugriff übersetzt
- ◆ Memory-Objekte sind nicht Teil des Objekt-Heaps
- ◆ Sharing von Memory-Objekten zwischen Domains möglich
  - ➤ Übergabe grosser Datenmengen zwischen Betriebssystem-Modulen und Anwendung (Netzwerk, Dateisystem, ...)

5 Optimierungen: Memory Objekte (2)



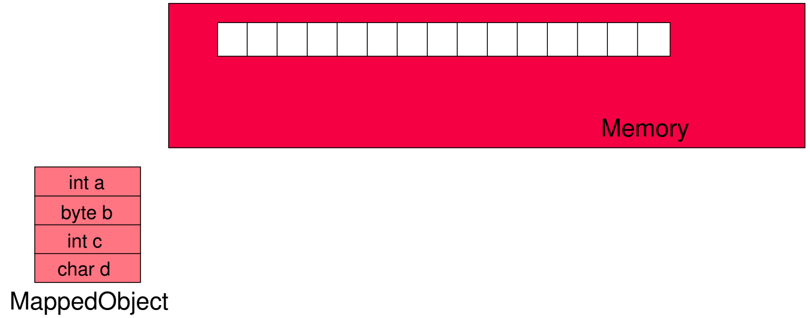
5 Optimierungen: Memory Objekte (3)

- Performance-Problem: Bereichsüberprüfung
- (teilweise) Lösung: Abbilden von speziellen Objekten auf Memory-Objekte (Mapping, Reifikation)



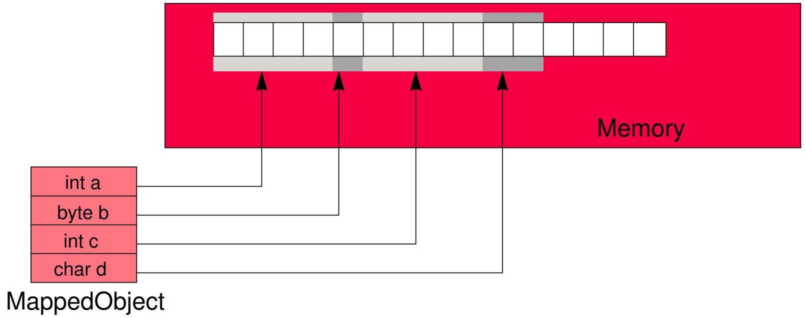
5 Optimierungen: Memory Objekte (4)

- Performance-Problem: Bereichsüberprüfung
- (teilweise) Lösung: Abbilden von speziellen Objekten auf Memory-Objekte (Mapping, Reifikation)

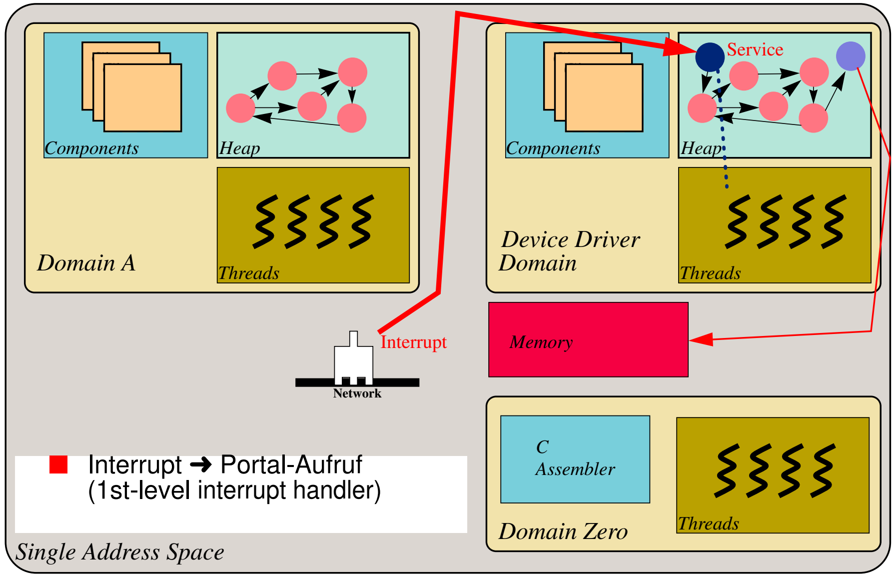


5 Optimierungen: Memory Objekte (5)

- Performance-Problem: Bereichsüberprüfung
- (teilweise) Lösung: Abbilden von speziellen Objekten auf Memory-Objekte (Mapping, Reifikation)



6 Gerätetreiber

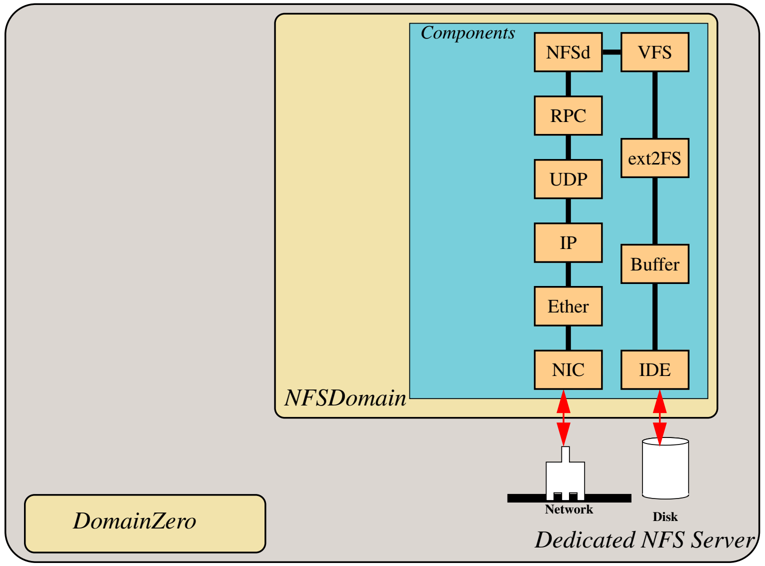


- Interrupt ➜ Portal-Aufruf (1st-level interrupt handler)

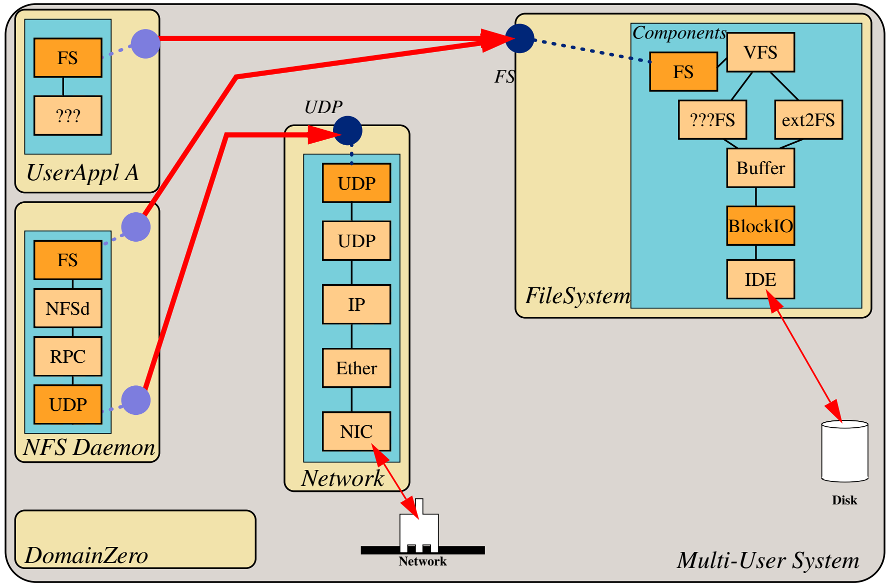
6 Gerätetreiber: Schutzproblematik

- Interrupts sind während der Ausführung des Interrupt-Handlers gesperrt
- Lösung:
  - ◆ Interrupt-Handler-Methoden müssen spezielles Interface implementieren
  - ◆ Verifier überprüft solche Methoden auf maximale Ausführungszeit
    - ➤ statische Analyse soweit möglich
    - ➤ Einfügen von Laufzeitüberprüfungen wo Ausführungszeit nicht entschiedbar
    - ➤ Laufzeitüberprüfungen können Interruptbehandlung abbrechen und Massnahmen gegen weiteres Fehlverhalten treffen (z. B. Interrupts von Gerät deaktivieren)
  - ◆ Aufteilen der Interrupt-Bearbeitung
    - ➤ kleiner vertrauenswürdiger 1st-level-Teil bearbeitet Initialisierung und DMA
    - ➤ komplexe Bearbeitung in JX-kontrolliertem 2nd-level-Teil

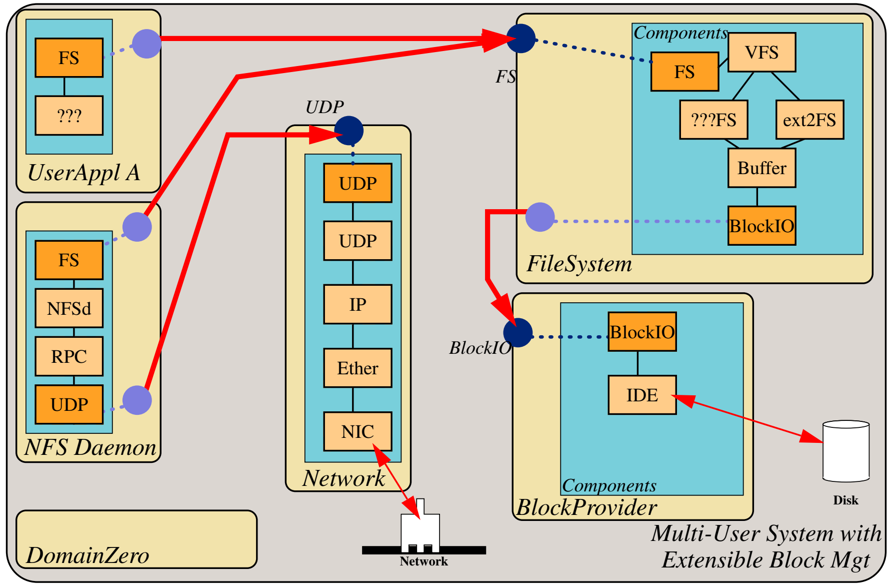
7 Betriebssystembaukasten: Spezialsystem



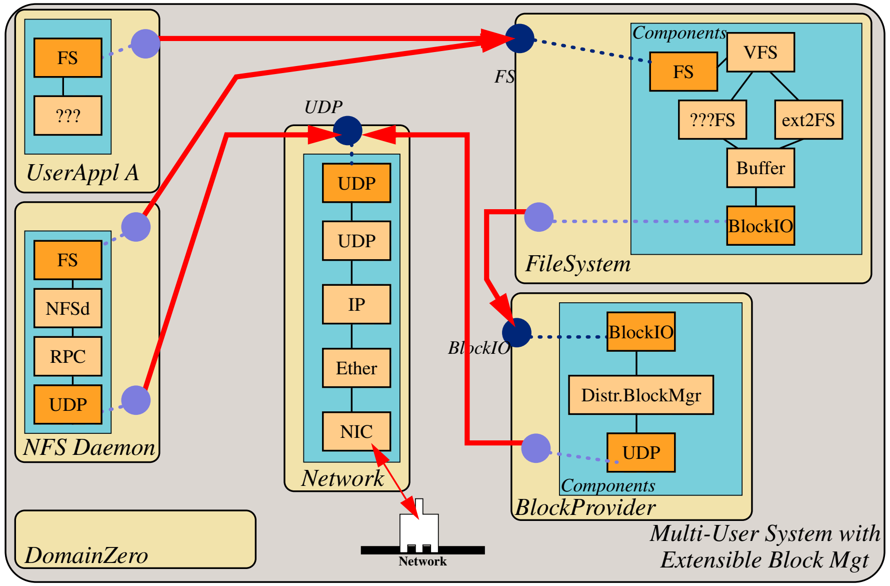
7 Betriebssystembaukasten: Multiuser-System



7 Betriebssystembaukasten: Erweiterbarkeit



7 Betriebssystembaukasten: Erweiterbarkeit (2)



8 Erweiterungsmöglichkeiten

- Betriebssystemmechanismen pro Domain
  - Scheduler
  - Garbage Collector
  - Compiler (aber: Sicherheitsproblem!)
- Erweiterter Portalaufruf
  - über Adressraumgrenzen
  - über Rechnergrenzen
    - ➥ verteiltes Betriebssystem
- "Smart Portals"
  - ganz oder teilweise replizierte Remote-Objekte (Fragment-Konzept)