Entwicklung der Betriebssystemstrukturierung

- Monolithische Kerne
  - Entstanden nach den ersten Batch-Systemen Anfang der 60er Jahre
  - Unangefochtenes Konzept bis Mitte der 80er Jahre
  - Beispiele: MVS, UNIX, VMS, BS2000
- Minimalkerne
  - Erste Ideen in der ersten Hälfte der 80er Jahre
  - Erste produktreife Systeme seit 1989/1990
  - Beispiele: MACH, Amoeba, (Windows-) NT, Chorus
- Objektbasierte, offene Betriebssysteme
  - Forschung seit Mitte der 80er Jahre (siehe auch [HSE91])
  - Mehrere lauffähige Prototypen als Forschungsergebnisse
  - Bislang keine Produkte auf dem Markt
  - Beispiele: Choices, Clouds [DLA+91], APERTOS [Yok92], Spring [HaK93]

Literatur

BAL+90 B. N. Bershad, T. E. Anderson, E. D. Lazowska and Henry M. Levy, “Lightweight Remote Procedure Call”, *ACM Transactions on Computer Systems*, Vol. 8, No. 1, pp. 37-55, Feb. 1990.

Dei90 H. M. Deitel, *Operating Systems*, 2nd Ed., Addison-Wesley, Reading (MA), 1990

DLA+91 P. Dasgupta, R. J. LeBlanc, M. Ahamed and U. Ramachandran, “The CLOUDS Distributed Operating System”, *IEEE Computer*, Apr. 1991.

HaK93 G. Hamilton and P. Kougiouris, "The Spring nucleus: A micokernel for objects", Proceedings of the *Summer 1993 USENIX Technical Conference*, pp. 147 - 159, Cincinnati (OH, USA), Jun. 1993.

HSE91 F. Hofmann, P. Schlenk, T. Eirich, “Encapsulation and Interaction in Future Operating Systems”, *Workshop on Operating Systems for the 90s and Beyond*, pp. 241-246, Dagstuhl (Germany), Eds. A. Karshmer and J. Nehmer, Lecture Notes in Computer Science No. 563, Springer-Verlag, 1991.

Laz93 E. Lazowska, "Recent Trends in Experimental Operating Systems Research", *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*, pp. 13 - 19, Ithaka (NY, USA), Aug. 1993.

Tan92 Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall Inc., Englewood-Cliffs (NJ), 1992.

Yok92 Yasuhiko Yokote, “The Apertos Reflective Operating System: The Concept and Its Implementation”, *OOPSLA ’92 – Conference Proceedings,* pp. 414-434, Vancouver (BC, Canada), published as *SIGPLAN Notices*, Vol. 27, No. 10, Oct. 1992.

Monolithische Betriebssystemkerne

❑ Alle Betriebssystemkomponenten zu einem Kern zusammengefaßt

- Programmierung
  - bis Anfang der 70er Jahre ausschließlich in Assembler
  - beginnend mit UNIX wesentliche Teile auch in höheren Programmiersprachen (Vor allem C, aber auch Sprachen, die von Rechnerherstellern explizit für die Betriebssystemprogrammierung entwickelt wurden — z. B. Cybel für das System NOS/VE auf der Cyber.)
- Bindung
  - statisch: alle Betriebssystemkomponenten werden zusammengebunden — der Kern wird "am Stück" gebootet
  - dynamisch: beim Booten wird nur ein Teil der Betriebssystemkomponenten geladen, weitere Moduln können während der Laufzeit dynamisch nachgeladen werden (z. B. im Sun-Betriebssystem Solaris)
- Kapselung
  - Alle Betriebssystemfunktionen sind in einem virtuellen Adreßraum zusammengefaßt

❑ Ausführung generell im privilegierten Modus

- Benutzerschnittstelle
  - SVC (Supervisor-Call) = spezieller Maschinenbefehl, der einen Wechsel des PSW bewirkt. Dadurch kontrollierter Einsprung in das Betriebssystem. Anschließend Umschalten der Schutzumgebung (virtueller Adreßraum) auf den Adreßraum des Betriebssystems.

Vorteile

+ Effiziente Kommunikation zwischen den Komponenten des Kerns durch Prozeduraufrufe
  - Da alle Betriebssystemfunktionen in einem Adreßraum zusammengefaßt sind, können sie sich gegenseitig direkt adressieren (= normaler Unterprogrammaufruf)

+ Direkter Zugriff auf alle Datenstrukturen des Kerns jederzeit möglich
  - Kern-Datenstrukturen können einfach als globale Daten angelegt werden und sind damit aus allen Unterprogrammen heraus direkt zugreifbar → sehr effizient

+ Privilegierte Befehle jederzeit aufrufbar
  - Da alle Kern-Funktionen im Supervisor-Modus ausgeführt werden, können privilegierte Befehle jederzeit ausgeführt werden und geschützte Adreßbereiche und Register (z. B. von E/A-Geräten) direkt angesprochen werden

Monolithische Betriebssystemkerne (2)

Nachteile

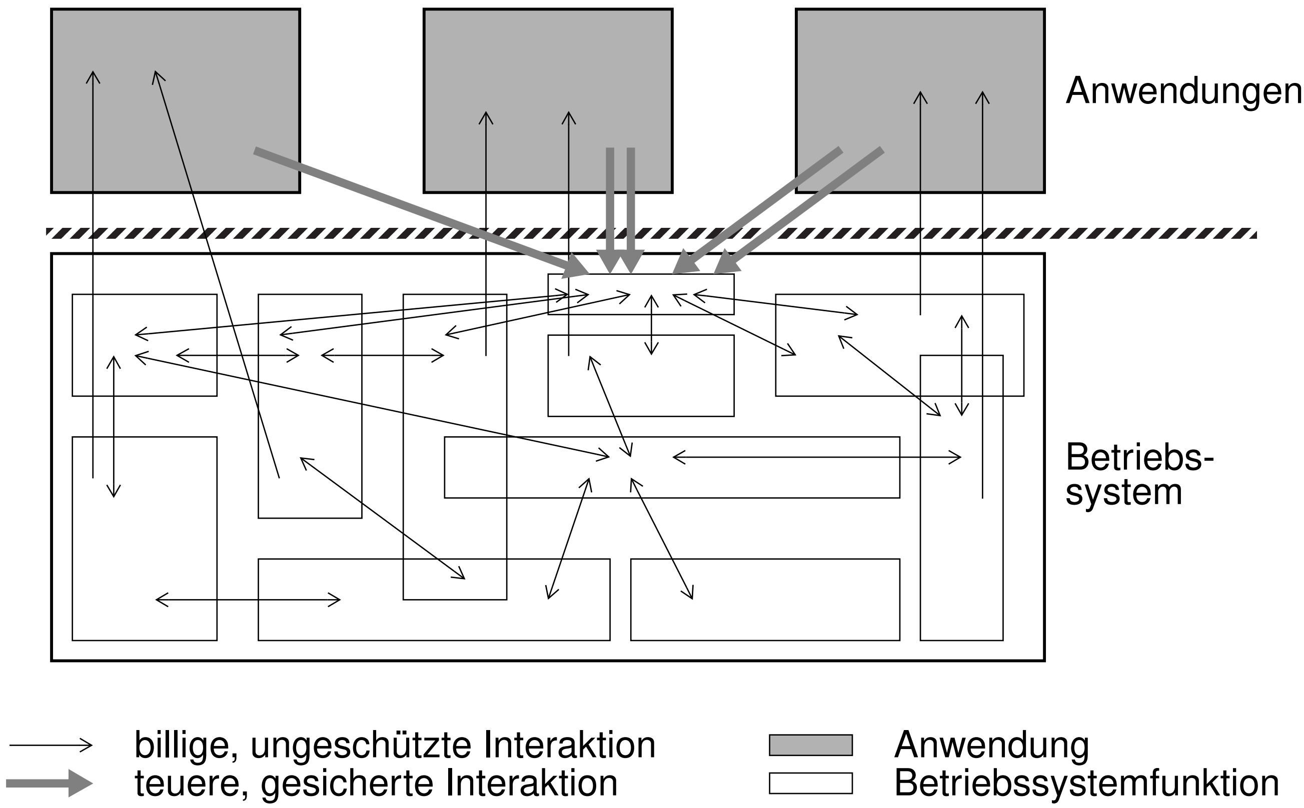
- Mechanismen strikt vorgegeben; individuelle Anpassung an Anwendungen, Erweiterung oder Reduzierung nicht möglich
  - Da die meisten monolithischen Betriebssystemkerne statisch gebunden sind, ist zur Laufzeit keine Änderung möglich.
  - Selbst wenn ein System ein Nachladen von Modulen ermöglicht, sind die Beeinflussungsmöglichkeiten nicht sehr groß: in der Regel können nur zusätzliche Treiber oder Teile, die über sehr flexibel konfigurierbare Schnittstellen innerhalb des Betriebssystems angesprochen werden (z. B. Dateisystem), nachgeladen oder sogar ausgetauscht werden.

- Strukturierung meist schlecht → änderungsunfreundlich und fehleranfällig
  - Durch die Programmierung in Assembler oder in maschinennahen höheren Programmiersprachen war bei den meisten Betriebssystemen kein Zwang zu guter Strukturierung und strenger Modularisierung vorgegeben.
  - Hohe Effizienzanforderungen haben häufig dazu geführt, daß intensiv Seiteneffekte ausgenutzt wurden — insbesondere durch direkten Zugriff auf zentrale Datenstrukturen des Systems. Bei Änderungen ist es daher oft schwierig, alle Stellen im Betriebssystem zu finden, die von dieser Änderung betroffen sind.

- Kein Schutz gegenüber Kernkomponenten möglich (kritisch z. B. bei zugekauften Treibern!)
  - Da alle Betriebssystemfunktionen im privilegierten Modus ausgeführt werden, gibt es keinerlei Schutzmechanismen.
    - Durch Programmierfehler können dadurch leicht andere Systembereiche zerstört werden → Systemabsturz.
    - Während Anwenderprogramme bei den meisten Systemen im nicht-privilegierten Benutzer-Modus ausgeführt werden und damit den Schutzmechanismen des Betriebssystems unterworfen sind (wodurch z. B. Virenprobleme wie bei MS-DOS weitgehend vermieden werden), kann sich das Betriebssystem nicht gegen Software schützen, die zu dem Betriebssystem hinzugebunden wurde (z. B. ein Treiber für ein spezielles E/A-Gerät). Solche Software kann potentiell beliebigen Schaden anrichten (wie Viren in ungeschützten Umgebungen wie MS-DOS).

- Niedriges Abstraktionsniveau
  Programmierung “direkt auf der nackten Hardware”
  - Anwendungsprogramme bauen auf den Abstraktionen des Betriebssystems (z. B. Dateien, Prozesse, Signale) auf. Der Programmierkomfort wird durch diese vom Betriebssystem bereitgestellten Abstraktionen erheblich verbessert.
  - Bei der Programmierung des Betriebssystems selbst stehen solche Abstraktionen natürlich noch nicht zur Verfügung — einzige Basis sind die Maschinenbefehle der Hardware. Zugriffe auf Betriebssystemabstraktionen können höchstens durch direkte Manipulation der entsprechenden Kern-Datenstrukturen erfolgen.

Monolithische Betriebssystemkerne (3)



- Anwendungen erreichen den Systemkern nur über eine zentrale Einsprung-Stelle (trap-Handler oder SVC-Handler)
  - hierbei muß auf den virtuellen Adreßraum des Betriebssystems umgeschaltet werden (und nach Ende des Systemaufrufs wieder zurück auf den virtuellen Adreßraum der Anwendung) — dadurch relativ teuer
- Anwendungen können untereinander nur mit Hilfe von Betriebssystemmechanismen kommunizieren — müssen also immer durch das Betriebssystem hindurch
- Betriebssystemfunktionen können sich direkt aufrufen (weil im gleichen virtuellen Adreßraum)
- Betriebssystemfunktionen haben direkten Zugriff auf die Daten von Anwendungen (und nutzen diesen auch, um z. B. Daten zwischen Anwendungen zu übertragen oder Ergebnisse von Betriebssystemdiensten zurückzuliefern — beispielsweise die Zeichen, die im Rahmen eines read-Systemaufrufs gelesen wurden)
  - hierbei muß normalerweise nicht der virtuelle Adreßraum umgeschaltet werden (da das Betriebssystem im privilegierten Modus ausgeführt wird, stehen entweder Maschinenbefehle zur Verfügung, die einen direkten Zugriff auf den physischen Hauptspeicher ermöglichen oder die benötigten Datenbereiche können in den virtuellen Adreßraum des Betriebssystems abgebildet werden)
    → billige Datenübertragung

Minimalkerne

❏ Auslagerung von Betriebssystemkomponenten

- Viele monolithische Kerne sind im Laufe jahrelanger Erweiterungen um sehr viele Funktionen ergänzt worden, die zwar einfach im Kern implementierbar sind, eigentlich aber nicht im privilegierten Modus ausgeführt werden müßten.
- Beispiele:
  - Netzwerkkommunikationsprotokolle wie TCP/IP
  - Editorfunktionen *(Line-Editing)* im Terminaltreiber
  - unterschiedliche Dateisystemimplementierungen

➔ Plazierung der Komponenten in eigenen Adreßräumen

- Betriebssystemfunktionsgruppen (z. B. ein Dateisystem) werden zusammengefaßt und aus dem Systemkern ausgelagert.
- Systemaufrufe, die von solch einer Funktionsgruppe auszuführen sind, werden von Anwendungen direkt an die ausgelagerte Funktionsgruppe gerichtet (als normale Interprozeßkommunikation).

➔ Ausführung im Benutzermodus

- Solange eine ausgelagerte Funktionsgruppe keine privilegierten Befehle ausführen bzw. auf privilegierte Register (z. B. Controller-E/A-Register) zugreifen muß, ist es nicht notwendig, sie im Supervisor-Modus auszuführen — sie kann wie eine "ganz normale Anwendung" gestartet werden.

❏ Reduktion des Kerns auf Basisfunktionalität und die Mechanismen, für die privilegierter Modus notwendig ist

- Um auf einem Minimalkern vollständige Betriebssystemmechanismen aufzubauen, reichen im wesentlichen die folgenden Funktionen aus:
  - Interprozeßkommunikationsunterstützung
  - einfache Speicherverwaltung
  - Einfache Prozeßverwaltungs- und Schedulingmechanismen
  - grundlegende E/A-Funktionen

Vorteile

\+ Bessere Modularisierung

\+ Ausgelagerte Teile leichter individuell zu gestalten

➔ zusätzlich angepaßte Module erzeugbar

- Es kann beispielsweise ein zusätzliches Dateisystemmodul gestartet werden, das spezielle Funktionalität für eine Anwendung bereitstellt.

➔ nicht benötigte Module können einfach weggelassen werden

\+ Ausgelagerte Teile werden nicht privilegiert ausgeführt → mehr Sicherheit

Minimalkerne (2)

Nachteile

– Kommunikation zwischen Modulen teuer
(RPC, 20- bis 70-facher Aufwand eines Prozeduraufrufs)
  - Mechanismen, die bisher innerhalb des Systemkerns in Form eines Prozeduraufrufs angesprochen werden konnten, müssen jetzt über Adreßraumgrenzen hinweg aufgerufen werden (= Interprozeßkommunikation oder Remote Procedure Call).
    - es werden Adreßraumumschaltungen notwendig (teuer, da die TLBs — Translation Lookaside Buffer, Cache der MMU für Einträge der Seitenkacheltabelle — ausgewechselt werden müssen)
    - an der University of Washington wurden Untersuchungen zur Optimierung von RPCs durchgeführt [BAL+90] — bei sehr intensiver Optimierung betrugen die Kosten für einen adreßraumübergreifenden Prozeduraufruf immer noch das 20-fache eines lokalen Prozeduraufrufs.

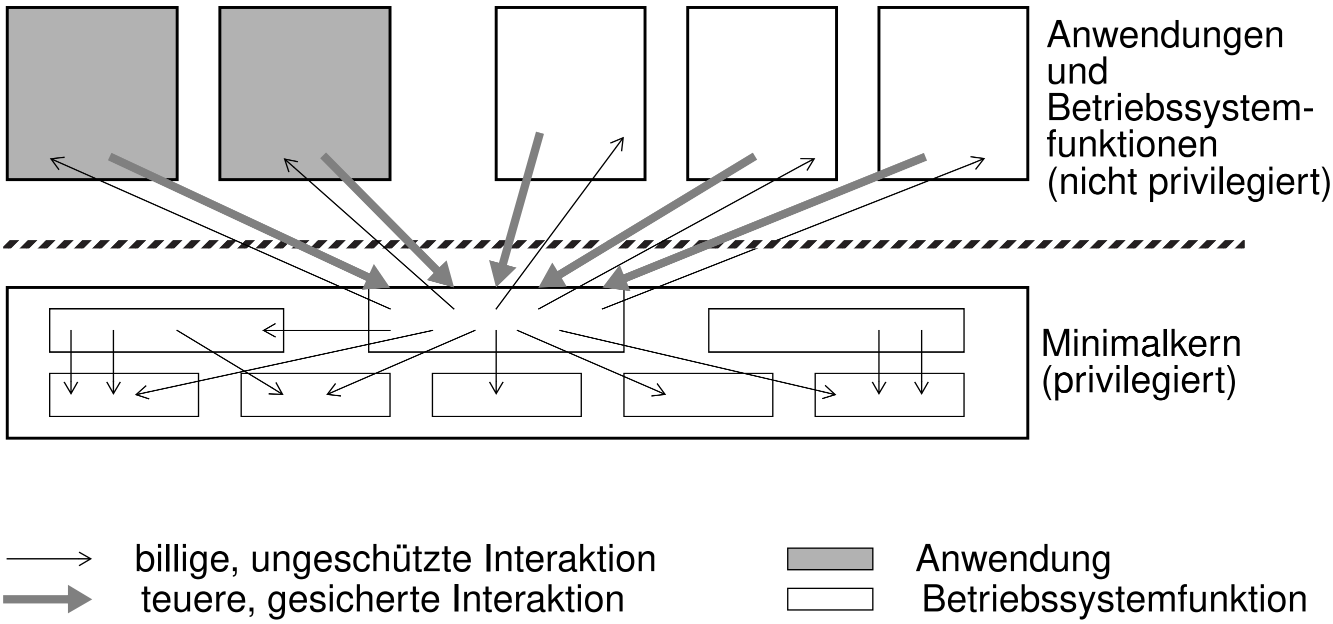
➜ Module werden aus Effizienzgründen möglichst groß gehalten
  - Wenn zwischen zwei Betriebssystemkomponenten sehr intensive Kommunikation stattfindet, ist es aus Effizienzgründen nicht vertretbar, die Komponenten in separate Adreßräume zu plazieren.

➜ innerhalb der Module Strukturierung beliebig
  - Wenn unterschiedliche Funktionsgruppen aus Effizienzgründen zu einem Modul zusammengefaßt werden, liegt es nahe, innerhalb dieses Moduls wieder mit gemeinsamen Datenstrukturen, Seiteneffekten, etc. zu arbeiten — man erhält die gleichen Probleme wie in einem monolithischen Kern.

➜ aus Effizienzgründen verbleiben Komponenten im Minimalkern, die nicht dorthin gehören
  - Komponenten, die intensiv von Funktionen des Minimalkerns genutzt werden, werden — um Adreßraumumschaltungen zu vermeiden — zwangsläufig wieder zu dem Minimalkern dazugebunden werden müssen.
    - der Minimalkern wird größer, als er eigentlich sein müßte
  - Innerhalb des Minimalkerns ist ebenfalls kein Zwang zu guter Modularisierung vorgegeben, die Nutzung gemeinsamer Datenstrukturen und die Bildung von Seiteneffekten wird nicht unterbunden
  - Durch die Auslagerung wesentlicher Kernkomponenten wie
    - Dateisystem
    - Netzwerkkommunikation
    - Prozeßverwaltung

    wird der Systemkern zwar verkleinert, die grundsätzlichen Probleme sind aber unverändert.

Minimalkerne (3)



- Die zentrale Funktion des Minimalkerns ist die Bereitstellung von Interprozeßkommunikationsmechanismen.
- Kommunikation zwischen Anwendungen sowie zwischen Anwendungen und Kernkomponenten funktioniert gleich — immer unter Vermittlung des IPC-Moduls im Minimalkern.

Objektbasierte, offene Betriebssysteme

❑ Fortführung des Wegs 'monolithischer Kern → Minimalkern'

- Der Übergang vom monolithischen Kern zu den Minimalkernen führte zu einer groben Modularisierung des Betriebssystems.
- Minimalkern und ausgelagerte Betriebssystemmodule sind jeweils gekapselt, Interaktion zwischen diesen Komponenten ist nur über die dafür definierten Schnittstellen (RPC oder *message passing*) möglich.
- Aus Effizienzgründen kann in einer Minimalkernarchitektur nur eine grobe Modularisierung vorgenommen werden, wobei zu beachten ist, daß intensive Interaktionen nicht über Modulgrenzen hinweg erfolgen.

❑ Weitere Modularisierung nur möglich, wenn Inter-Modul-Kommunikationskosten reduziert werden

➔ Adreßraumkapselung muß aufgegeben werden

- Wenn unterschiedliche Module in einem Adreßraum zusammengelegt werden, ist eine Inter-Modul-Kommunikation nicht wesentlich teurer als ein normaler Prozeduraufruf.

➔ Verlust an Sicherheit

- Es wird nun nicht mehr durch die Hardware (MMU) verhindert, daß ein Modul unberechtigt auf Daten anderer Module zugreift.
- Fehler in einem Modul können damit wieder wie bei einem monolithische Kern das Gesamtsystem in Mitleidenschaft ziehen.

➔ Sicherheit muß anders gewährleistet werden

➔ Sprachebene + objektorientierte bzw. objektbasierte Programmierung

- Die Programmiersprache erlaubt Interaktionen zwischen Modulen nur über definierte Schnittstellen und überprüft, daß die Module keine anderen Zugriffe auf die "Außenwelt" durchführen.
- Die Beziehungen zwischen Modulen werden damit konsequent auf die Modulschnittstellen beschränkt — bei Änderungen oder Erweiterungen des Gesamtsystems wird dadurch die Gefahr unbeabsichtigter Nebenwirkungen reduziert.

❑ Wenn Sicherheit auf Sprachebene garantiert wird, kann auf Adreßraumgrenzen verzichtet werden, sonst nicht!

- kann für ein Modul "Wohlverhalten" gengenüber anderen Modulen nicht garantiert werden, kann es aus Sicherheitsgründen in einen anderen Adreßraum "eingesperrt" werden.

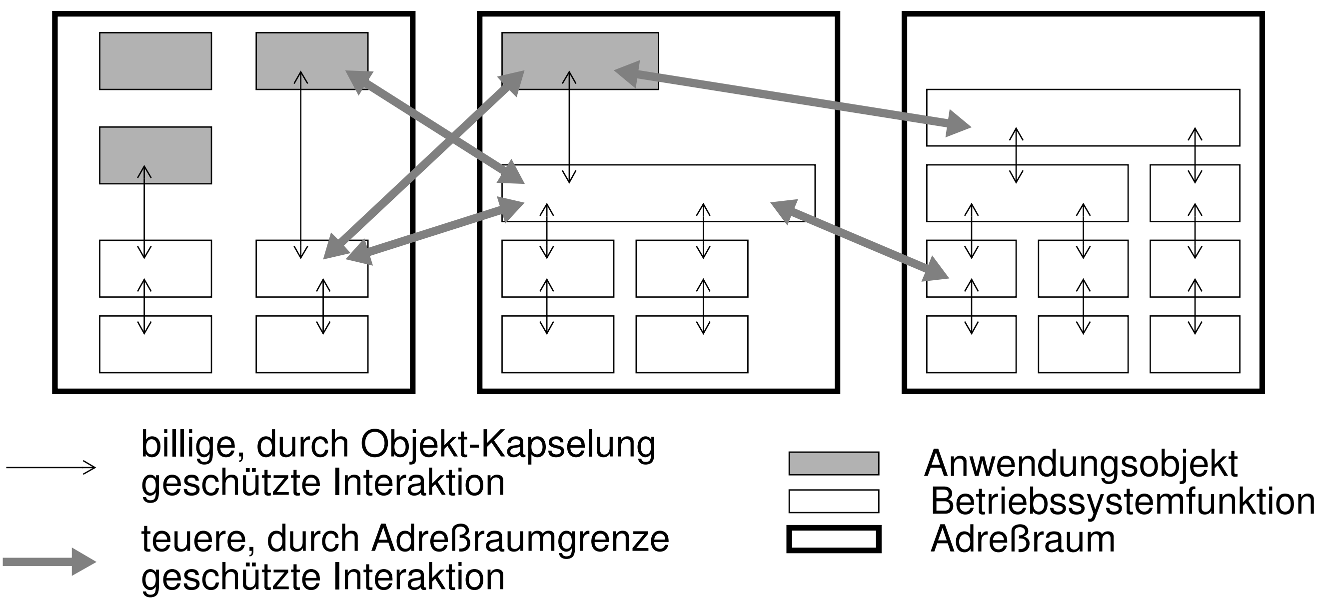
➔ gleiche Modul-Interaktion Adreßraum-lokal und Adreßraum-übergreifend wünschenswert

- Bei einer einheitlichen Schnittstelle für lokale und Adreßraum-übergreifende Modulinteraktion (d. h. syntaktische und semantische Übereinstimmung!) muß der Programmierer die Fälle nicht mehr explizit unterscheiden.
- Module können auch später ohne Programmänderungen noch verlagert werden.

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
  - Die bislang verbreiteten objektorientierten Programmiersprachen und deren Laufzeitsysteme konnten diese Sicherheitsgarantien nicht leisten (Hauptproblem: Zeiger!). Mit Java steht seit kurzem aber tatsächlich eine Sprache zur Verfügung, deren Laufzeitsystem die erforderlichen Sicherheitsüberprüfungen durchführen kann, so daß gefahrlos verschiedene Anwendungen in einem Adreßraum ausgeführt werden können (Ziel: single-adress-space-system)



3-BS-Architekturen.doc: 24.06.97 15:25

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 numerische 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.