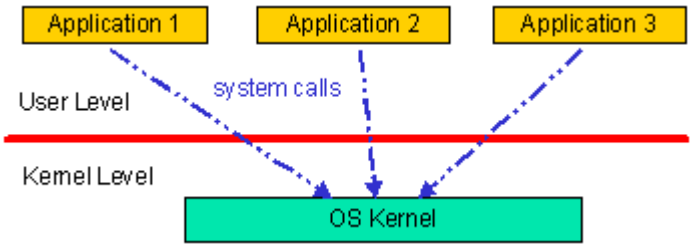
Kapitel 3: Überblick über Systeme

3.1 Motivation und Einführung

Eine wesentliche Aufgabe eines Betriebssystems (OS) ist es, quasi als vermittelnde Instanz zwischen Anwendern und der Computerhardware zu wirken und dabei so von den Hardwaredetails zu abstrahieren, dass Benutzer und Anwendungsprogrammierer leichter Anwendersysteme (*application systems*) implementieren und pflegen können. Darüber hinaus fungiert ein Betriebssystem auch als Manager von Systemressourcen (*resources*), welche sowohl Hardware- als auch Software-Objekte wie z.B. eine Datei sein können. Drittens agiert ein Betriebssystem auch als Kontrolleur, der sowohl das Zusammenspiel von Anwender- und Systemprogrammen als auch das Vorankommen der Anwenderprogramme (*applications*) überwacht, eventuell sogar deren Voranschreiten mittels Monitorprogramme beobachtet und gegebenenfalls einschränkt. Viertens übernimmt ein Betriebssystem auch die Aufgabe, einen Computer sicher und zuverlässig laufen zu lassen. Schlussendlich sollte ein Betriebssystem auch Instrumente zur Verfügung stellen, mit deren Hilfe die Dienstleistungen des Rechners abgerechnet werden können.

Aus den Erfahrungen einer ca. 50 jährigen Betriebssystementwicklung sollte eine moderne Systemarchitektur ihre Lehren ziehen, um so einer Reihe von Herausforderungen besser als in der Vergangenheit begegnen zu können. Die essentielle gegenwärtige Herausforderung zumindest bei den uns vertrauten Betriebssystemen für PC, Workstations und Server scheint uns die Beherrschung der Komplexität zu sein. Um zu neuen Lösungen zu kommen, sollten wir jedoch auch die traditionellen Lösungsansätze betrachten, um deren Stärken, aber auch deren Schwächen zu erkennen. Andererseits sollten wir auch nicht unterschlagen, dass die Mehrzahl der heute eingesetzten Betriebssysteme aus dem Konsum- bzw. Echtzeitsystembereich kommen. Für all diese höchst unterschiedlichen Systemanforderungen allgemein gültige Entwurfsprinzipien, nützliche Verfahren, Konzepte etc. zu finden, ist das primäre Lehrziel dieser Veranstaltung. Beispielhaft werden wir uns jedoch auch möglichen Lösungen für Spezialsysteme nähern, um so den vollen Gestaltungsspielraum von Systemarchitekten zumindest ansatzweise auszuloten.

Seit Brinch Hansen besteht der traditionelle Architekturansatz für Betriebssysteme darin, wesentliche Systemfunktionen in einem Systemkern (*kernel*) zusammenzufassen, der in einem privilegierten Prozessormodus abläuft, d.h. ihm stehen i.d.R. alle Befehle (instructions) des Prozessors zur Verfügung, während im Anwenderbereich (*user level*) nur ein eingeschränkter Befehlssatz zur Verfügung steht. Das folgende Bild vermittelt einen Überblick über ein kernorientiertes Betriebssystem, wobei Anwendungen mittels Systemaufrufe (*system calls*) über die so genannte Systemschnittstelle (*API = application programmer interface*) Systemdienste, die im Systemkern realisiert sind, aufrufen können.



Diese Zweiteilung beim Systementwurf wird von den meisten heutigen Rechnerarchitekturen auch hardwaremäßig unterstützt, indem Prozessoren einerseits im Anwendermodus (*user mode*) und andererseits im Systemmodus (*kernel mode*) tätig sein können. Ein privilegierter Befehl in einem Anwenderprogrammteil kann somit von der Hardware entdeckt werden, i.d.R. führt dies zu einer Ausnahme(*exception*). Ausnahmen unterbrechen die aktuelle Befehlsausführung und können so genutzt werden, um auf diese Ausnahmesituationen mittels Ausnahmebehandlungen (*exception handlers*) zu reagieren.

Fragen:

1. Wozu dient in erster Linie diese Aufteilung von Systemkern und Anwendungen bzw. Systemfunktionen, die auf Anwenderebene durchgeführt werden?

2. Versuchen Sie die Schwächen im traditionellen kernorientierten Systemansatz zu eruieren und entwerfen Sie entsprechende verbesserte Systemarchitekturen.

3.2 Allgemeine Ziele von Betriebssystemen

In der folgenden Aufzählung werden die in der Literatur häufig genannten Systemanforderungen aufgeführt. Einige dieser Entwurfsziele können auch direkt übertragen werden, wenn es um Entwurf und Implementierung eines Anwendersystems (*application system*) geht.

Bequemlichkeit (*Convenience*)
: Erleichtert Programmieren (bzw. Nutzung) von Anwendungen (*facilitate programming of applications*)

Abstraktionen und Standards (*abstraction and standardization*)
: Verbirgt die Details der Hardware (*hide hardware details*)
: Ermöglicht eine einheitliche Schnittstelle für verschiedene E/A-Geräte (*provide uniform interface for different I/O devices*)

Allgemeingültigkeit (*Generality*)
: Veränderungen in der Charakteristik der Hauptanwendungen sollten keinen kompletten Neuentwurf des Systems nach sich ziehen (*changes in the characteristics of major applications should not require a complete redesign of the OS*)

Kundenorientierung (*customizablity*)
: Anpassbarkeit auf Anwenderprofile (*adapt to specific applications' requirements*)

Erweiterbarkeit (*Extensibility*)
: Ermöglicht die Entwicklung, das Austesten und die Einführung von neuen Systemfunktionen ohne das aktuelle System zu beeinträchtigen (*permit development, testing and introduction of new system functions without interfering with current service*)

Skalierbarkeit (*Scalability*)
: System soll in der Lage sein, auch mit einer wachsenden Anwenderlast oder zusätzlichen Hardwarekomponenten fertig zu werden (*system should be able to face ever increasing load conditions and additional hardware components*)

Effizienz (*Efficiency*)
: System sollte möglichst schnell sein und dabei auch seine Betriebsmittel (*resources*) gut und möglichst Strom sparend ausnutzen (*make good use of resources, be fast, and consume few power*)

Dienstgüte (*Quality of Service*)
: System sollte Mindestanforderungen der Benutzer garantieren können (*guarantee certain amount of system service*)

Robustheit (*Robustness*)
: Kein Systemabsturz, nur weil eine Anwendung fehlerhaft oder böswillig falsch angelegt worden ist (*No system breakdown due to a malicious or erroneous application*)

Sicherheit und Schutz (*Security and Protection*)
: Anbieten von Fairness, Sicherheit und Schutz (*Should provide fairness, security, and safety*)

Fragen:

1. Können Sie für jedes konkrete Entwurfsziel ein ganz konkretes System- oder Anwenderbeispiel nennen? Diskutieren Sie schon mal vorab, wie man diese Ziele geschickt umsetzen könnte. Überlegen Sie sich hierzu Analogien aus dem täglichen Leben und überlegen Sie, ob die dort gefundenen Lösungen auf die Systemarchitektur übertragbar sind.

2. In welchem Zusammenhang sind Ihnen ähnliche Zielvorgaben in Ihrem bisherigen Studium schon in der einen oder anderen Veranstaltung begegnet?

3.3 Job eines Systemarchitekten

1. Nützliche Systeme zu entwickeln (*design useful and customizable systems*)

a. Wie kann man messen, dass ein System nützlich ist? Antwort: Ein nützliches System wird oft gekauft! Zieht dieses Argument immer?
b. Gibt es andere Nützlichkeitskriterien? Wenn ja, welche?

2. Strukturiere Systeme so, dass sie leicht erweiterbar sind und wartbar bleiben (*design extensible and maintainable systems*)
   a. Zukünftige PC-Anwendersysteme werden im Laufe der Systembenutzung mit großer Wahrscheinlichkeit eher mehr denn weniger Ressourcen benötigen und u.U. auch neue Systemfunktionalitäten in Anspruch nehmen.
   b. Die Systempflege wird bei manchen kommerziellen Systemen zu einer Art Alptraum, wenn man sie durchzuführen hat oder wenn man als Kunde mit *Update-Patches* überfrachtet wird (siehe den letzten XP security patch).

3. Integriere in das System nicht funktionale Eigenschaften, z.B. Robustheit (d.h. geringere Fehleranfälligkeit) und Qualitätsgarantien (*establish non-functional properties, e.g. robustnss and quality-of-service guarantees (QoS)*)

4. Implementiere letztendlich korrekte, sichere und robuste Systeme (*implement correct, secure, and robust systems*)

3.4 Betriebssystemhistorie

Es gibt eine Reihe von Artikeln und auch Büchern über die Entwicklung von Softwaresystemen im Allgemeinen und von Betriebssystemen im Besonderen. Wer sich darüber näher informieren möchte, dem sei folgender Link empfohlen: http://www.osdata.com/kind/history.htm

3.5 Systementwurf

Wie ein Systementwurf **softwaretechnisch** mit den geeigneten Hilfsmitteln (***tools***) in der Praxis durchgeführt werden sollte, ist Gegenstand der parallel stattfindenden gleich lautenden Wahlpflichtveranstaltung "Softwaretechnik".

3.5.1 Entwurfskriterien

Beim Entwurf eines Betriebssystems können eine Reihe von einflussreichen Kriterien unterschieden werden, deren Nichtbeachtung weit reichende negative Folgen haben kann. Beim Entwurf eines Systems für einen bestimmten Spielcomputer (*z.B. gameboy*) sind sicher andere Schwerpunkte als bei einem Supercomputer (*z.B. IBM's DataStar rated at 10.4 teraflops*) zu setzen. Während beim *Gameboy* ein niedriger Stromverbrauch essentiell ist, spielt(e) dies bei einem Supercomputer à la *Cray-1* eine eher untergeordnete Rolle.

Folgende Entwurfsparameter können zumindest teilweise unabhängig von einander gesehen werden, spannen somit also einen **8-dimesionalen orthogonalen Entwurfsraum** auf:

1. Systemgröße
   - Allerweltssysteme (*pervasive systems, e.g.* Multimediaplayer, Haushaltsroboter, etc.)
   - Eingebettete Systeme (*embedded systems, e.g. production lines, control systems*)
   - Handhelds, NCs
   - PCs, Workstations
   - Mainframes
   - Supercomputer
   - Cluster
   - LAN, MAN, WAN

2. Preis
   - Billigprodukte

    - Gehobene Preisklasse, ...

3. Leistung
    - Langsam getaktete, ...
    - Super schnelle Rechner

4. Skalierbarkeit
    - Nicht skalierbare Systeme
    - Schwach skalierbare Systeme
    - Hoch skalierbare Systeme

5. Vielseitigkeit
    - Dedizierte Rechner für dedizierte Anwendungen
    - Allzwecksysteme

6. Sicherheit
    - Offene ungeschützte Systeme
    - Entsprechende Sicherheitsklassen (*e.g. according to the Orange Book*)

7. Homogenität
    - Homogene Systemkomponenten
    - Heterogene Systemkomponenten

8. Mobilität
    - Stationäre Systeme
    - ...
    - Mobile Systeme

Natürlich sind nicht alle Ausprägungen pro Kriterium mit jeder Ausprägung eines anderen Kriteriums beliebig sinnvoll kombinierbar, beispielsweise wird es auf absehbare Zeit keine mobilen Supercomputer geben, auch wenn in einigen Jahren mobile Systeme nahe an die Leistungsfähigkeit heutiger Supercomputer kommen mögen. Allerdings wird die "Unersättlichkeit von uns Menschen" dann auch wieder neue Supersupercomputer benötigen, so wie heutige Desktoprechner beispielsweise nahe an die Leistung von Supercomputern der 80iger Jahre heranreichen.

3.5.2 Abstraktionen und Systemkonzepte

Eine Reihe von Abstraktionen bzw. Konzepten haben sich im Verlauf der Betriebssystementwicklung als nützlich herausgestellt. Einige davon sind zur Einstimmung aufgeführt, werden aber in den späteren Kapiteln noch näher untersucht werden.

3.5.2.1 Prozess, Task, Thread und Adressraum

Damit Anwendungen (*applications*) auf einem Betriebssystem kontrolliert und geschützt ablaufen können, also ebenda einen Fortschritt (*lat. procedere*) erzielen können, benötigen wir einen Begriff, der diese überwachte Tätigkeit charakterisiert. Wir sprechen von einem Prozess, wenn eine Anwendung aus einem geschützten Adressraum (Analogon: Haushalt) besteht, in dem genau nur eine mögliche Aktivität (*activity*) (Analogon eine Person) vorgesehen ist (Analogon also ein Singlehaushalt). Die Mehrzahl aller in den ersten beiden Semestern geschriebenen Programme beinhaltet genau dieses streng sequentielle Ablaufschema. Würde man jedoch auf diese Weise ein Schachprogramm entwerfen, dann wäre vermutlich seinerzeit Gary Kasparov beim Rematch zwischen Mensch und Maschine nicht von *Deep Blue* besiegt worden.

Stellen wir uns zur Veranschaulichung eine Anwendung vor, die gemäß einem Suchkriterium in einem riesigen Datenvolumen (z.B. in einer sequentiell organisierten, aber ungeordneten Datei) nach einem ganz bestimmten Datensatz sucht. Führt man diese Suche streng sequentiell aus und selbst wenn man das Suchen pro Datensatz (inklusive Einlagern in den Hauptspeicher) in nur einer µsec durchführen könnte, so benötigt man im Mittel bereits 12 Stunden, wenn insgesamt "nur" 86 400 000 000 Suchelemente vorhanden sind. Man

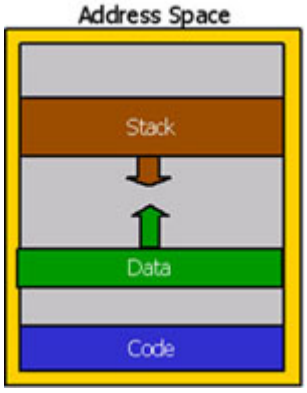
mag auf den ersten Blick solche Datenlager als astronomisch und unrealistisch betrachten, jedoch sind solche Zahlen in der Tat nicht nur auf Anwendungen in der Astronomie beschränkt, sondern betreffen ebenfalls den Mikrokosmos.

Wenn es uns also bei obiger Anwendung gelänge, sowohl das Durchsuchen der Datensätze als auch deren Einlagerung in den Hauptspeicher zu parallelisieren, so könnte man beispielsweise die Rechenzeit je nach erreichbaren Parallelitätsgrad entsprechend reduzieren. Überlegen Sie sich hierzu ein geeignetes Programmschema in einer Programmiersprache Ihrer Wahl.

Eine ganz andere Frage ist es, wie man den Parallelitätsgrad einer Anwendung nicht nur de facto sondern wirklich erhöhen kann. Offensichtlich können wir echte Parallelität dann gewinnen, wenn wir ein Mehrprozessorsystem, z.B. ein SMP-System benutzen können. Beispielsweise könnte im Idealfall obige Anwendung auf einem 4-Prozessorsystem somit schon in 3 Stunden und auf einem 720-Mehrprozessorsystem sogar schon nach einer Minute bearbeitet sein. Hat man jedoch ein Einprozessorsystem vor sich, dann ergibt sich auf den ersten Blick keine Notwendigkeit für eine parallelisierte Anwendung. Wir werden jedoch im weiteren Verlauf der Vorlesung erarbeiten, dass auch in einem Einprozessorsystem quasiparallele Anwendungen in Form nebenläufiger Aktivitäten, durch so genannte parallele Aktivitäten entweder traditionell als parallele Prozesse oder neuerdings auch als parallele Threads gewinnbringend eingesetzt werden können.

Manche Systeme bzw. Programmiersprachen bieten deswegen neben dem klassischen Prozessschema auch dieses modernere Threadschema an, bei dem pro Adressraum bis zu n>1 konkurrierende Aktivitäten (*threads*) bestimmt werden können (Analogon: Familienhaushalt oder Wohngemeinschaft). Ein Thread steht hierbei für die Einheit, die als separate, eigenständige Aktivität im ganzen System oder zumindest in einem Subsystem angesprochen werden kann. Wir wollen im Folgenden einen Adressraum mit mehr als einem Thread als Task bezeichnen.

Tasks oder Prozesse sind i.d.R. die Einheiten, denen ein Betriebssystem Ressourcen (z.B. Hauptspeicher oder Dateien) zuordnet, allerdings besitzt jeder Thread seinen eigenen Stapel (*stack*). Manche Anwendungen verwenden das Prinzip der gemeinsamen Codebenutzung (*codesharing*) bei den Threads einer Task (z.B. bietet sich das im obigen Beispiel an), andere Anwendungen benötigen pro Thread einen eigenen Code, der aber nicht notwendigerweise in separat voneinander geschützten Adressraumbereichen implementiert sein muss.



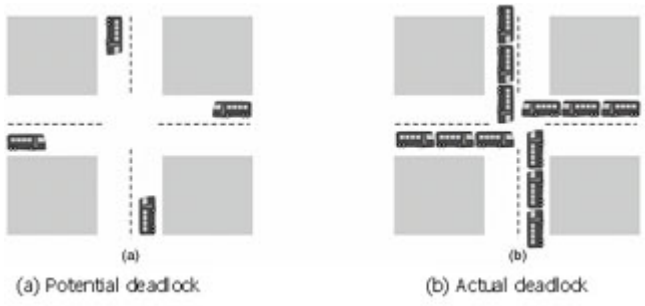
Der obige Unix ähnliche Prozessadressraum enthält genau drei Adressraumbereiche (manchmal auch "logische" Segmente genannt), einen für den Code, einen für globale Daten (inklusive Heap) und einen für die temporären Daten auf dem Stapel (*stack*). Die grau hinterlegten Abschnitte dieses Adressraums sind gegenwärtig nicht benutzt bzw. nicht benutzbar. Sollte also in einer Anwendung auf Grund eines Adressierungsfehlers (z.B. nicht initialisierte Zeigervariable in C) ein Speicherwort des grauen Bereichs gelesen oder beschrieben werden, dann verursacht die Hardware wiederum eine Ausnahme (exception), vorausgesetzt man hat der Hardware beigebracht, dass diese grauen Abschnitte nicht zu diesem Adressraum gehören sollen. Dies ist im Übrigen die Aufgabe des Betriebssystemkerns.

Frage: Überlegen Sie sich für diese grauen Bereiche eine Analogie zu unserem Haushaltsbeispiel.

3.5.2.2 Konkurrenz oder Nebenläufigkeit

Da wir nun echt parallele oder auch nur quasi parallele Aktivitäten, eben die Threads in das Spiel gebracht haben, handelt man sich dadurch auch eine Reihe von Konkurrenzproblemen (*race conditions*) ein, für welche der Systemarchitekt dem Benutzer leicht verständliche und robuste Konzepte und Mechanismen anbieten muss, so dass der Anwenderprogrammierer diese Konkurrenzprobleme möglichst vermeiden kann. Im Bereich der Betriebssysteme wird deswegen häufig das so genannte Verklemmungsproblem (*deadlock problem*) als gravierend betrachtet, obwohl einige kommerzielle Systeme sich überhaupt nicht um dessen Lösung kümmern mit der etwas scheinheiligen Begründung, dass Verklemmungen auch im tägliche Leben nicht allzu häufig vorkommen (siehe das unten angegeben Beispiels aus dem täglichen Straßenverkehr).

Beispiele:



Bei einer Verklemmung (*deadlock*) blockieren sich mindestens zwei Prozesses oder Threads gegenseitig, wobei z.B. der erste Prozess eine Ressource R1 zuerst anfordert und auch vom System zugeteilt bekommt, während der zweite Prozess zunächst die Ressource R2 anfordert und ebenfalls bekommt. Wenn nun der erste Prozess zusätzlich noch R2 benötigt und der zweite Prozess zusätzlich noch R1 benötigt, dann warten beide Prozesse ohne Zusatzmaßnahmen ewig aufeinander, d.h. keiner der beiden wird jemals wieder einen Fortschritt erzielen können.

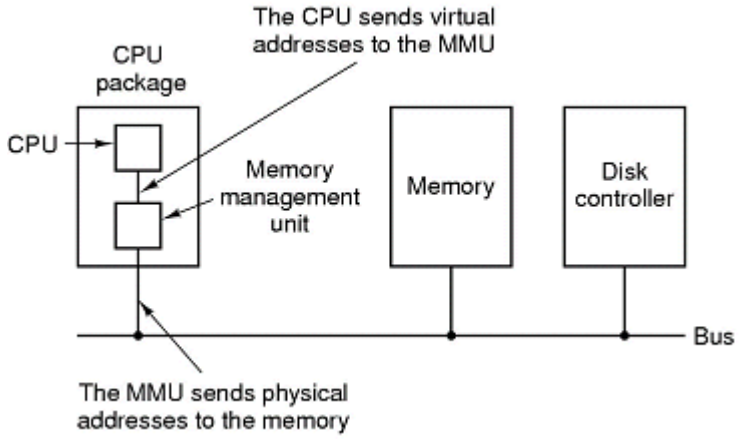
Andere Beispiele für das Parallelitätsproblem sind Wettbewerbssituationen an gemeinsam zugreifbaren Daten (z.B. globale Daten im Adressraum oder Dateien). Wenn wiederum mindestens zwei Threads gleichzeitig auf globale Daten verändernd zugreifen, dann kann es leicht zu inkonsistenten Daten kommen.

3.5.2.3 Speicherverwaltung und virtueller Speicher

Manche Betriebssystemleute argumentieren: *"Bei heute gängigen Hauptspeichergrößen könnte man dieses Thema eigentlich übergehen. Man kann jeder Anwendung genügend Hauptspeicher permanent zuordnen."*

Dies ist jedoch nur bedingt richtig, da Anwendungen immer komfortabler implementiert werden, u.a. mit graphischen Benutzeroberflächen, in Zukunft wahrscheinlich noch zusätzlich mit Audiointeraktivität (*headphone* und *microphone*), so dass Speicherverwaltung sehr schnell wieder zu einem spannenden Thema werden wird. Außerdem gibt es auch Systeme, in denen man aus anderen Gründen (z.B. mangelnder Platz oder wenig Energieverbrauch) auf umfangreichen Hauptspeicher verzichten möchte.

Beim virtuellen Speicherkonzept unterscheidet man logische (oder virtuelle) Adressen und physische oder reale Adressen, die auf bestimmte Stellen im Hauptspeicher verweisen. Damit man das virtuelle Speicherkonzept anwenden kann, muss die Hardware eine Abbildungseinheit anbieten, die zur Laufzeit aus jeder logischen Adresse möglichst schnell die reale oder physische Adresse ermittelt. Diese Hardwareeinheit heißt MMU (= *memory management unit*, eine nicht ganz so glückliche Notation).



Frage: Welche der im Kapitel 2 angesprochenen Hardwareeinheiten sind bei einem virtuellen Speicherkonzept effektiv notwendig und welche erhöhen u.U. die Effizienz eines Systems.

3.5.2.4 Ressourcenverwaltung und Scheduling

Man kann die übrigen peripheren Hardwarekomponenten, also Drucker, Netzwerkkarten etc. nach rein kommerziellen Aspekten an die Prozesse und Tasks verteilen, d.h. man versucht alle Systemressourcen möglichst effizient auszulasten, oder man implementiert eine eher nutzerfreundliche Ressourcenverwaltung, bei der man auch auf die individuellen Bedürfnisse der verschiedenen Anwendungen Rücksicht nimmt. Insbesondere manche Mehrbenutzersysteme werden versuchen, alle aktuellen aktiven Benutzer möglichst fair zu behandeln, indem sie pro Benutzer einen gleich großen Ressourcenanteil (zumindest einen gleichen Prozessoranteil) vorsehen werden. Insgesamt bedarf es somit ausgeklügelter Planungsstrategien (*scheduling policies*), wie die limitierten Systemressourcen auf die Benutzer bzw. Anwendungen aufgeteilt werden sollen. (So wenig wie man Mitarbeiter ständig Überstunden machen lassen sollte, so wenig sollte man Systeme überlasten, im letzten Fall kann sogar schon eine kurzfristige Systemüberlastung zu extremen Leistungseinbußen (*system thrashing*) führen.)

3.5.2.5 Interprozesskommunikation

Gelegentlich müssen Anwendungen auch miteinander interagieren, wenn sie beispielsweise an einer gemeinsamen Problemlösung zusammenarbeiten. Wenn dabei die Anwendungen so implementiert sind, dass sie aus Schutzgründen in getrennten Adressräumen untergebracht sind (also beispielsweise als sequentielle Prozesse), dann muss das System dafür sorgen, dass Information zwischen diesen voreinander geschützten Adressräumen fließen kann. Wie wird dies im täglichen Haushalt durchgeführt? Auch hierbei gibt es prinzipiell zwei verschiedene Lösungsansätze:

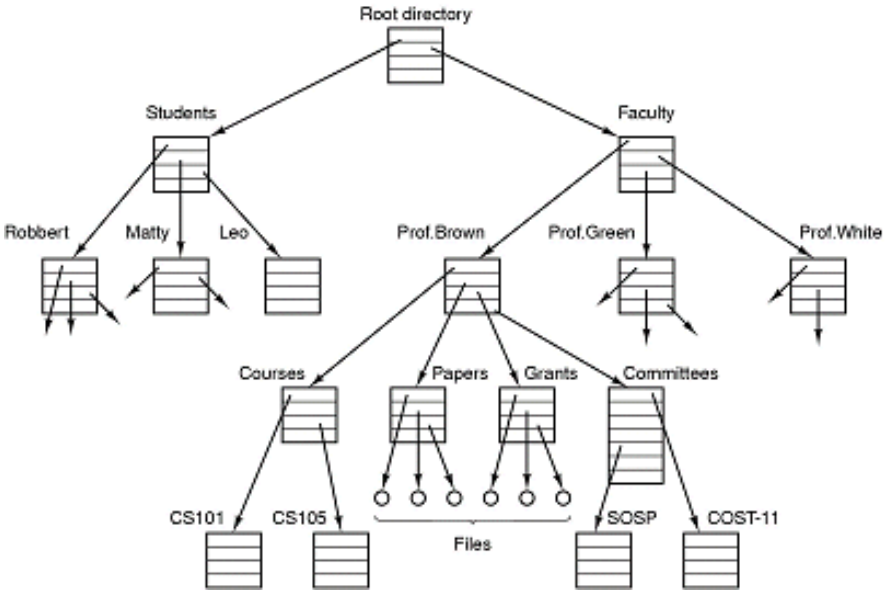
1. Man setzt sich ans Telefon und bespricht das weitere gemeinsame Vorgehen.

2. Es gibt im Wohnblock einen Gemeinschaftsraum, in dem man sich treffen und absprechen kann.

Genau diese beiden Gestaltungsprinzipien der Interaktion gibt es auch in Systemen, im ersten Fall spricht man von gerichteter Kommunikation, bei der ein Datentransport von einem Adressraum zum nächsten durchgeführt wird, im zweiten Fall spricht am von "gemeinsamen Speicher" (*shared memory*), d.h. also einem Adressbereich, der zu mehreren Adressräumen gehört.

3.5.2.6 Dateisystem (File System)

Dateisysteme dienen der langfristigen Informationsspeicherung, d.h. sie müssen auf einem nicht flüchtigen Speichermedium implementiert werden, wie z.B. Magnetband (*tape*), Magnetplatte (*disk*), CD-ROM, Diskette etc.



Während im legendären MS-DOS noch ein so genanntes flaches Dateisystem implementiert worden ist, bieten mittlerweile viele kommerzielle Systeme ein Unix ähnliches Dateisystem an, in dem über Ordner oder Verzeichnisse leichter und bequemer auf Dateien zugegriffen werden kann. Neben dem üblichen traditionellen Dateizugriff über Dateideskriptor und Dateikopf (*inode*) kann man in einigen Systemen Dateien auch in den Adressraum einer Anwendung direkt abbilden (*memory mapped file*). Wenn man demzufolge deren Anfangs- und Endadresse kennt, kann man auf jedes Byte einer direkt abgebildeten Datei wie auf einen Vektor zugreifen.

Frage: Kennen Sie Dateianwendungen, in denen es keinen großen Nutzen bringt, direkt abgebildete Dateien zu verwenden?

3.5.2.7 Schutz und Sicherheit

Auf Sicherheit und Schutz wird in der Spezialveranstaltung: "Grundlagen der Computersicherheit" explizit eingegangen.

3.5.2.8 Unterstützende Funktionen, Werkzeuge (Tools)

Auf Anwenderebene existieren bei den meisten Systemen, auf denen Programme entwickelt werden sollen, zusätzliche Softwarewerkzeuge wie beispielsweise Editoren, Übersetzer (*compiler*), Assemblierer (*assembler*), Binder (*linker*) und Lader (*loader*). Ob man nun diese Werkzeuge mit zu einem Betriebssystem zählt oder nicht, das ist eher eine Geschmacksache, denn eines intensiven Diskurses wert. An der Mensch-/Maschinenschnittstelle sind heutzutage graphische Bedienungsoberflächen die Regel, während Sprachunterstützung wohl erst in der nächsten Systemgeneration im großen Stile anzutreffen sein wird. Obwohl es nach wie vor Programmiergurus geben wird, die nicht auf einen Kommandointerpretierer à la Unix (e.g. `bsh` oder `ksh`) verzichten wollen, so hat nicht zuletzt die GUI von Microsofts inklusive Plug and Play für den beachtlichen kommerziellen Erfolg der MS-Windowsbetriebssysteme gesorgt.

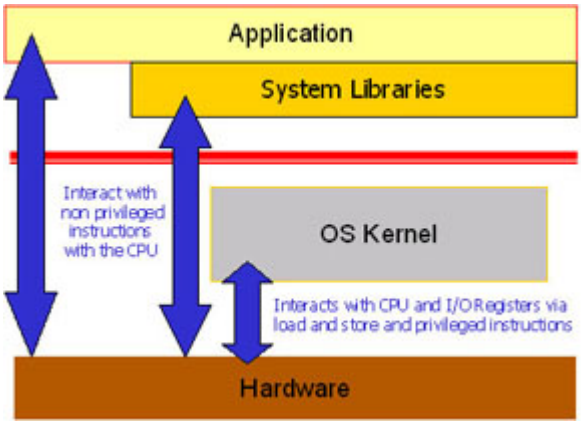
3.5.2.9 Kommandointerpretierer (Shell)

Ein Kommandointerpretierer stellt eine traditionelle Schnittstelle zwischen Benutzer und dem System dar. Altgediente Unix and Linux Gurus wollen i.d.R. nicht auf all die schönen Eigenschaften der Shell-Programmierung verzichten und sind nur schwer von den Vorteilen einer graphischen Benutzeroberfläche zu überzeugen.

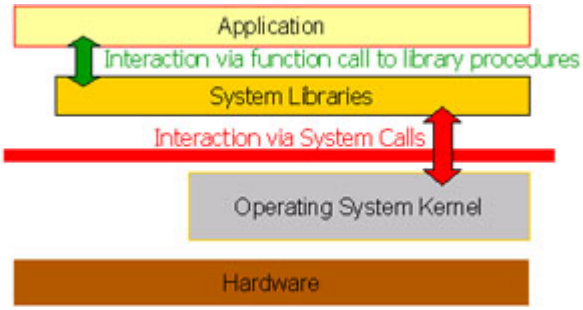
Mittlerweile haben aber diese graphischen Benutzeroberflächen die meisten PC-Benutzer überzeugt. GUI, Plug and play war MS's Antwort für alle enthusiastischen "Super-shellers". Andererseits ist nicht zu leugnen, dass viele extravagante Anwenderlösungen nur mittels Shellprogrammierung zu erreichen sind. Die

jeweiligen Extremgurus sollten also nicht nur entweder oder sagen, sondern wie häufig im Leben sowohl als auch; oder anders herum ausgedrückt, es schadet also auch einem modernen systemnah programmierenden Anwender nichts, wenn er einige der eleganten Features der Shellprogrammierung beherrscht.

3.5.2.10 Anwendung und Systemkern (OS-Kernel)



Ein Großteil der Aktionen auf dem Prozessor wird in den Anwendungen direkt mittels nicht privilegierter Befehle im Benutzermodus (*user mode*) ausgeführt. Bestimmte Aktionen benötigen jedoch auch privilegierte Befehle, für die in den meisten Systemen nur Funktionen des Kerns vorgesehen sind. In der Regel werden diese Kernfunktionen nicht direkt über die *System Call*-Schnittstelle, sondern über für Programmiersprachen spezifische Bibliotheksfunktionen aufgerufen, die ihrerseits die Kernfunktion aufrufen.



Pro System kann es mehrere solcher Bibliotheken geben (siehe auch Laufzeitsystem einer Sprache z.B. C Bibliothek).

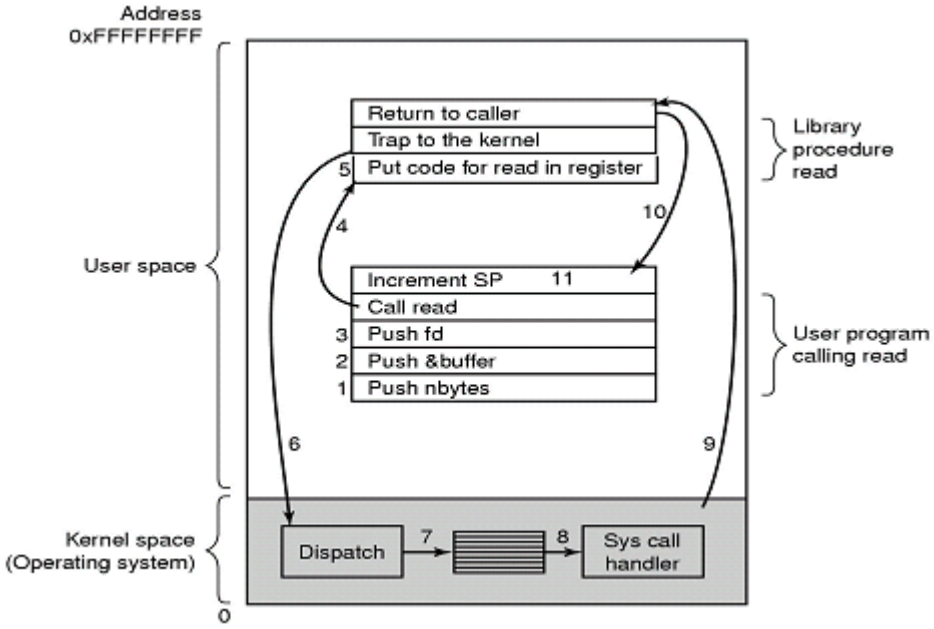
Man beachte jedoch, dass nur eine Teilmenge der Bibliotheksfunktionen einen Systemaufruf (*system call*) impliziert. Es gibt eine ganze Reihe häufig benutzter Bibliotheksfunktionen, die nicht auf die Hilfe des Systemkerns angewiesen sind, z.B. strcmp(), memcpy() sind typische User-Land Funktionen. Dagegen implizieren open(), read() System Calls Übergänge vom Benutzer- in den Kernmodus. Dieser Übergang ist bei manchen Rechnerarchitekturen mitunter recht teuer, d.h. kostet viele Taktzyklen, weswegen man auch versucht sein könnte, möglichst viele Systemfunktionen außerhalb des Kerns zu implementieren. Dabei wird häufig vergessen, dass man dann aber zu deren Nutzung über geeignete, d.h. höchst effiziente IPC-Mechanismen verfügen muss (die meisten IPC-Mechanismen beinhalten ihrerseits wiederum einen Übergang von Benutzer- in den Kernmodus).

Eine Reihe von Forschungssystemen (Exokernel, K42) benutzen Bibliotheken u.a. auch dafür, bisher im Kern angebotene Systemdienste ohne den aufwendigen Übergang vom Benutzermodus in den Kernmodus zu implementieren.

3.6 Systemstrukturen

Im traditionellen Systemarchitekturansatz werden Systemdienste fast ausnahmslos im privilegierten Systemkern implementiert, auch wenn sie keine privilegierten Prozessorbefehle benötigen. Um solch einen Dienst benutzen zu können, wird eine Anwendung direkt oder indirekt einen Systemaufruf (*system call*) absetzen, z.B. um aus einer geöffneten Datei Information im Umfang von b Bytes zu lesen.

```
count = read(fd, buffer, nbytes)
```



Legende zur obigen Abbildung (aus A. Tanenbaum: MOS).

Step 1-3: Caller pushes parameters on its user-stack (in reverse order). Note:
First and third parameter are of type: call by value,
Second parameter buffer is of type: call by reference.

Step 4: Call of the intermediate library function read.

Step 5: Library function puts system call number in a place where the OS-kernel expects it to be, e.g. in a specific register.

Step 6: Then executes TRAP-instruction in order to switch from user-mode to kernel-mode and starts executing at a fixed address within the kernel (No longer true for micro-kernel systems!!).

Step 7: Kernel examines system call number, if correct switches via interrupt vector table to start-address of the specified system call handler routine.

Step 8: The system call handler runs performing the read function.

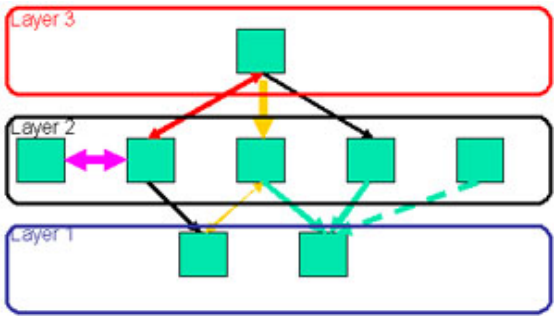
Step 9: Once, the system call handler has completed its job, control is given back to the library function at the instruction that follows the trap.

Step 10: This library function returns to the caller in the usual way procedures return.

Step 11: Upon return to the caller, its stack will be cleared, and it can use the desired buffers in &buffer

Hinweis: Ein Systemarchitekt sollte die POSIX-Schnittstelle studiert haben!

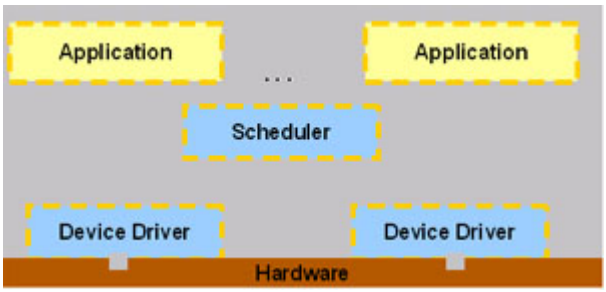
3.6.1 Geschichtete Systeme (Layered Systems)



Sinn und Zweck von geschichteten Systemen ist die bessere Beherrschung der Systemkomplexität, in dem man eine Ordnung zwischen den Schichten einführt und damit die Interaktionsmöglichkeiten zwischen Komponenten aus verschiedenen Schichten einschränkt, so dass beispielsweise nur Komponenten aus höheren Schichten Komponenten niederer Schichten aufrufen dürfen. Das erste Betriebssystem, das nach diesem Strukturierungsprinzip entworfen und implementiert wurde, war das "THE System" von Dijkstra, dem dann das MULTICS-System (entwickelt am MIT) folgte.

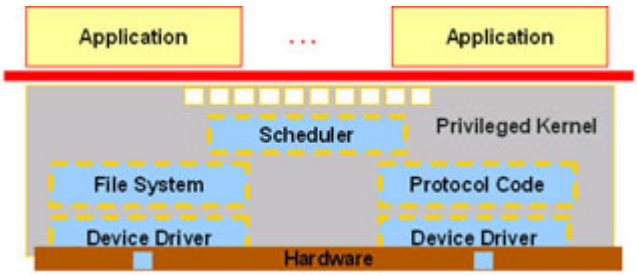
Frage: Aus welchen Gründen war MULTICS ein kommerzieller Erfolg verwehrt, obwohl dessen Entwurf in der Systemsgemeinde hohes Ansehen hatte?

3.6.2 Monolithische Systeme



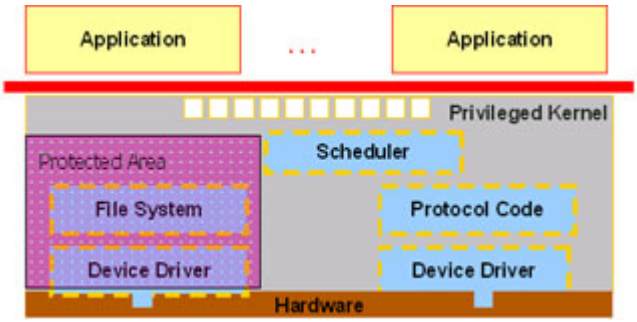
Monolithische Systeme zeichnen sich im wesentlichen dadurch aus, dass sie zwar sehr effizient programmiert werden können, da alle Programme den gleichen Adressraum teilen, aber dass sie nur sehr schwer weiterentwickelt und erweitert werden können, da jegliche Änderung am bestehenden System auch unerwünschte Seiteneffekte auf die Anwendungen haben können. Dieser Systemtyp wird allenfalls bei "Eingebetteten Systemen" (*embedded systems*) verwendet.

3.6.3 Monolithische Systemkernsysteme



Bewertung:
Die meisten Betriebssystemdienste werden innerhalb des privilegierten Kerns implementiert, d.h. somit ist der Kern vor den Anwendungen geschützt. Die Anwendungen sind wechselseitig geschützt, da sie in getrennten Adressräumen implementiert sind. Nach wie vor sind jedoch die Kernkomponenten untereinander nicht geschützt, ferner müssen Anwendungen dem Kern trauen.

3.6.4 Erweiterbare Kerne



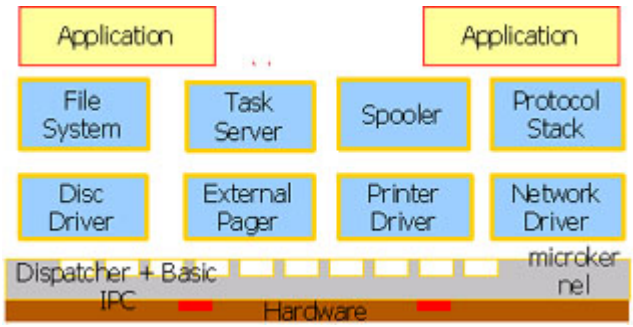
Zur Laufzeit können selten benötigte oder nachträglich implementierte Kernkomponenten (z.B. neue Treiber) in extra besonders geschützte Kernbereiche (*sand boxes*) geladen werden, wo sie dann solange ihre Dienste ausführen, wie sie von den Anwendungen benötigt werden.

3.6.5 Mikrokerne

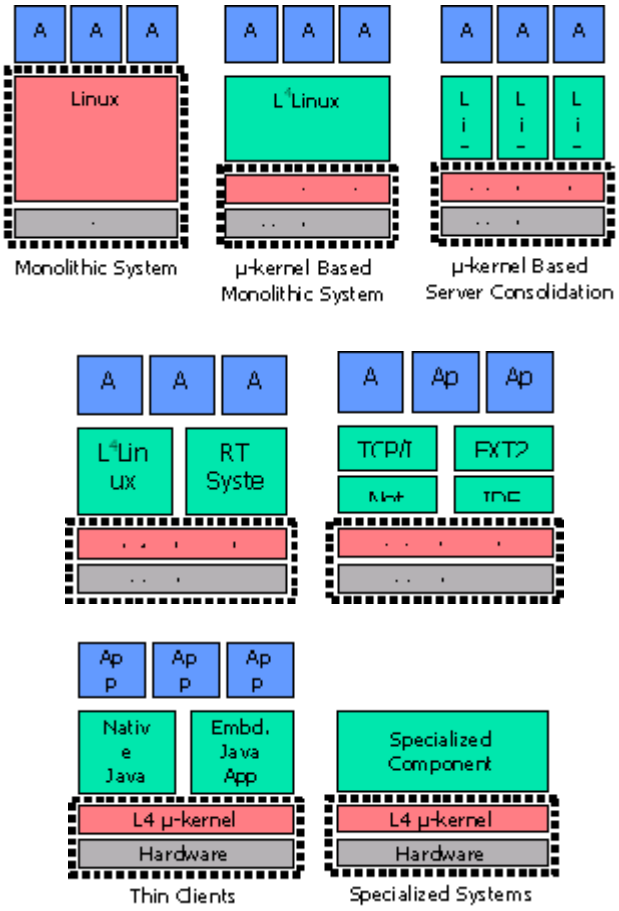
Der Mikrokernansatz ist nicht gänzlich neu (siehe u.a. CMU's MACH-Projekt 1985-1994), neu ist hingegen, dass dieser Ansatz nicht notwendigerweise in eine Leistungssackgasse führen muss. In einem Mikrokern werden nur die absolut notwendigen Abstraktionen implementiert und die hierzu notwendigen Infrastrukturmechanismen bereitgestellt. Im L4Ka Mikrokern **Pistachio** gibt es nur zwei essentielle Abstraktionen zusätzlich zu den beiden unten angegebenen Interaktionsmechanismen:

- Addressraum
- Thread
- Interprozesskommunikation (IPC)
- Abbilden von Adressbereichen (*map etc.*)

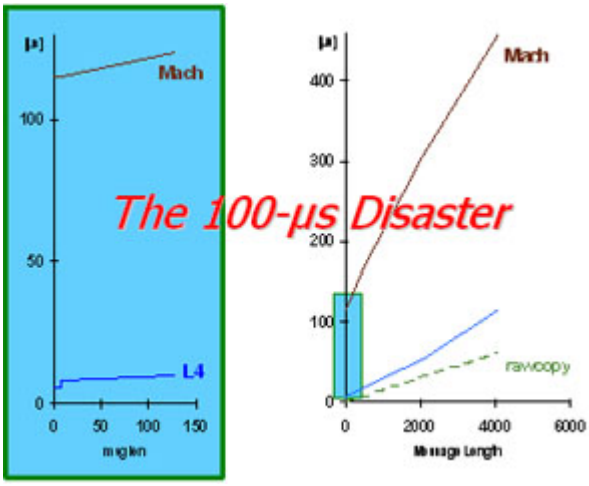
Alle anderen Systemdienste sollten als Dienstgeber (server) in getrennten Adressräumen außerhalb des Mikrokerns oder in Bibliotheken realisiert werden. Traditionelle Systemaufrufe werden auf IPC zwischen Anwendung und Server abgebildet.



Mögliche Anwendungen von Mikrokernen sind in der folgenden Abbildung angedeutet. Wer sich näher für den Aufbau und die Konstruktion von Mikrokernen interessiert ist, dem seien die entsprechende Vorlesung im Sommersemester und die Veröffentlichungen des Lehrstuhls empfohlen. In der Klausur wird u.a. auch erwartet, dass man sowohl die Vor- als auch die Nachteile von Mikrokernbasierten Systemen erläutern kann. Die meisten Nachteile sind auf falsche Entwurfskonzepte und ineffiziente Implementierungen derselben in den Mikrokernen der ersten Generation zurückzuführen. Nur wenige Mikrokerne der ersten Generation wurden überhaupt ihrer Namensgebung gerecht, sie waren einfach zu groß und komplex und widerspiegelten somit auch nur ungenügend die erhofften zusätzlichen Systemeigenschaften, wie Flexibilität etc.



In der folgenden Abbildung wird der IPC MACH mit dem von Jochen Liedtke entwickelten und in Assembler implementierten L4-Mikrokern verglichen. Bei kleinen Nachrichtenlängen von einigen wenigen Bytes war der Machkern um den Faktor 100 langsamer als der so genannte "Short-IPC" in L4, erstaunlicherweise ging dieser Vorsprung auch beim Übergang zu größeren Nachrichten nicht mehr verloren.



Da die Mikrokerne der ersten Generation nur höchst ineffiziente IPC (siehe oben) angeboten hatten, ergaben sich daraus auch nicht tolerierbare Folgekosten für Mikrokernbasierte, Server orientierte Betriebssysteme.

Frage: Diskutieren Sie die prinzipiellen Vor- und Nachteile von Mikrokern orientierten Systemen!

3.7 Verschiedene Systemtypen

In der Vorlesung wurden einige der folgenden Systemtypen angesprochen, der Fokus im weiteren Verlauf der Veranstaltung wird aber eindeutig auf Betriebssystemen für Ein- und Mehrprozessorsystemen liegen. Sie werden das häufigste Beispiel für Systemarchitekturkonzepte bleiben, wenn gleich hier noch einmal darauf verwiesen wird, dass es eine große Zahl von anderen Systemen gibt, die ähnliche oder gleiche Anforderungen an einen Systemarchitekten stellen.

- Datenbanksysteme
- Betriebssysteme
- Echtzeitsysteme
- Middleware
- Anwendungssysteme (z.B. Wettervorhersage)

3.8 Basisbegriffe (Basic Notions)

Die folgenden Begriffe sind zwar gebräuchlich, sind aber teilweise nicht sehr aussagekräftig, wie insgesamt viel zu viele Begriffe im Umfeld von Betriebssystemen eher schlampig und teilweise sogar sinnentstellend definiert worden sind.

- Single/Multi-Prozessor
- Single/Multi-Processing
- Single/Multi-Address Space
- Single/Multi-Programming
- Single/Multi-User

Frage: Geben Sie zu jedem der oben genannten Systemtypen ein ganz konkretes kommerzielles Beispiel an!