Konzepte von Betriebssystem-Komponenten

Aufbau eines Modernen Betriebssystems (Windows NT 5.0)

Moritz Mühlenthaler

14.6.2004

1a) Überblick über das Designproblem

Dass es heute eine Vielzahl von verschiedenen Betriebssystemen gibt lässt darauf schliessen, dass es kein Patentrezept für das Betriebssystemdesign gibt. Um besser zu verstehen, warum das so ist, muss man sich zunächst klar machen auf welche Probleme man im Allgemeinen bei der Entwicklung eines modernen Betriebssystems stösst.

- hohe Lebensdauer
  Verglichen mit "normalen" Programmen in der schnelllebigen Software-Welt erscheint die Lebensdauer von gängigen Betriebssystemen wie UNIX (ca 25 Jahre alt) und Windows (ca 10 Jahre alt) ungewöhnlich hoch. Natürlich bleibt die Entwicklung z.B. der Hardware in dieser Zeit nicht stehen, und ein gutes Betriebssystem muss sich den neuen Technologien anpassen können.

- Platformunabhängigkeit
  Damit ein Betriebssystem in mehreren Marktbereichen eine Rolle spielen kann, sollte es natürlich auf allen gängigen Platformen (Intel, Alpha, ...) laufen und sich an deren Eigenheiten anpassen können. Auch dass verschiedene HW-Komponenten von unterschiedlichen Herstellern unabhängig voneinander designt werden, und es die Aufgabe des Betriebssystems ist, dass diese Komponenten möglichst fehlerfrei zusammenarbeiten, macht es den BS-Entwicklern nicht gerade leicht.

- Komplexes Resourcen Management
  Es werden nicht nur verschiedenste Arten von Resourcen belegt/freigegeben/verändert, manche werden auch gemeinsam von unterschiedlichen Hardware-Komponenten benutzt (Interrupts, DMA-Kanäle, ...). Resourcen, die nicht "öffentlich" zur Verfügung stehen, müssen natürlich auch vor unautorisiertem Zugriff geschützt werden (private Dateien, Ports, ...)

- Abwärtskompatibilität
  Für Kunden ist es ärgerlich, wenn ihre alten Programme auf einer neuen ("vorteilhaften", "viel besseren") Betriebssystem-Version nicht mehr laufen. Deswegen ist es für die Entwickler sehr wichtig, dass ihr Betriebssystem abwärtskompatibel ist. Einschränkungen und Eigenarten von früheren Versionen müssen dann natürlich ebenfalls berücksichtigt werden (z.B. 8.3 Dateinamen bei Win95).

- Betriebssysteme sind sehr grosse Programme
  Wegen der Vielzahl von Diensten, die ein modernes Betriebssystem anbieten sollte, kommen diese mittlerweile auf astronomische Lines-of-Code- Zahlen, wobei UNIX mit 1-3 Mio Zeilen gegen das etwa 29 Mio Zeilen umfassende Windows 2000 eher bescheiden aussieht. Dieser Vergleich ist mit Vorsicht zu geniessen ist, da bei Windows 2000 natürlich Module wie das GDI dabei sind, die in UNIX nicht im Betriebssystem enthalten sind. Programme dieser Grössenordnung sind selbstverständlich nicht mehr im Ganzen zu überblicken bzw. zu verstehen.

1b) Design Goals

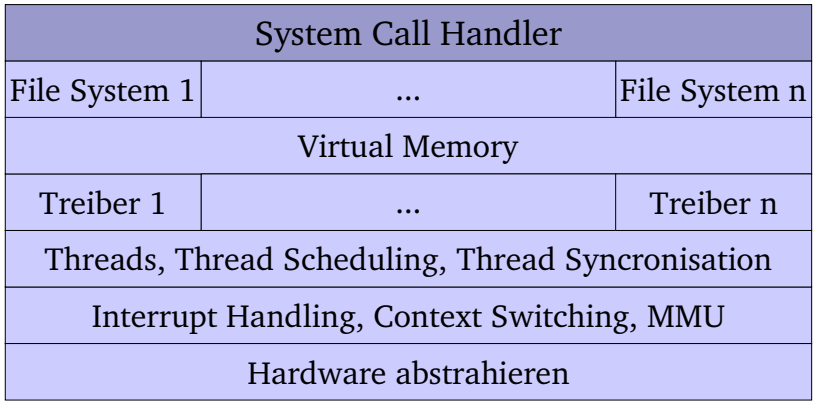
Nachdem wir eine kleine (nicht unbedingt repräsentative) Auswahl der Probleme betrachtet haben bleibt die Frage, ob man sich aus den Problemen nicht eine Liste von Zielen ableiten kann, die man bei der Entwicklung eines modernen Betriebssystems im Auge behalten sollte. Recht treffend erscheint die Aufzählung der Design Goals in [2].

- Erweiterbarkeit
- Portierbarkeit
- Zuverlässigkeit
- Kompatibilität
- Sicherheit
- Effizienz

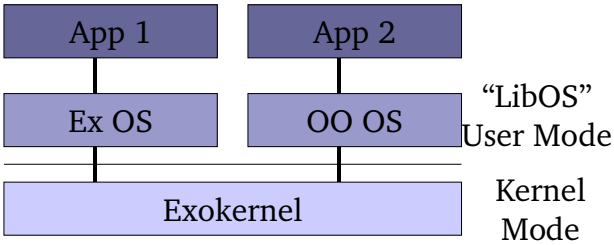
1c) Möglichkeiten der Strukturierung

Da, wie schon aufgezeigt, Betriebssysteme sehr grosse Programme sind, ist es fast unumgänglich das Problem 'Betriebssystemdesign' möglichst gut zu strukturieren, damit es überschaubarer wird. Ich möchte im Folgenden einige verbreitete Strukturierungs-Ansätze vorstellen.

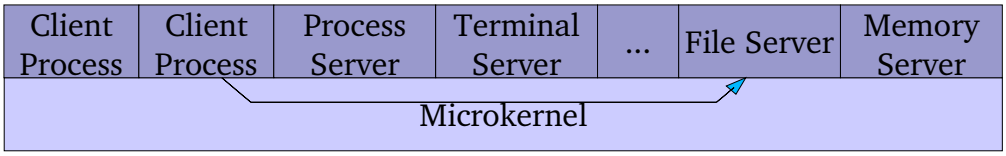
1. Schichten-Modell/Monolithischer Kernel:
   Eine sehr gängige Möglichkeit ein Betriebssystem zu strukturieren ist die Aufteilung in voneinander abgegrenzte Schichten, um einem monolithischen Kernel zumindest ein bischen Struktur zu geben. Die Aufgabe des Betriebssystem-Designers ist es, diese Aufteilung möglichst geschickt zu wählen. Da Systemdienste wie Treiber und Dateisysteme im Kernel Mode laufen, werden unnötige Context Switches gespart, was sich positiv auf die Effizienz des Systems auswirkt. Allerdings ist der Kernel als ganzes wegen seiner Grösse schwer zu überblicken.



2. Exokernel
   Bei der Exokernel Architektur gilt das zum Schichten-Modell gegensätzliche Prinzip: Es gibt einen Minimalkernel, der sich nur um das sichere Belegen und Freigeben von Resourcen kümmert. Alle Restlichen Dienste, wie z.B. das Dateisystem sind User Libraries, die den Anwendungen bei Bedarf zur Verfügung gestellt werden. Es gibt User Libraries, die z.B. den Kompletten Satz an UNIX Systemaufrufen zur Verfügung stellen (sog. “LibOS”).



3. Client-Server/Microkernel
   Ein Kompromiss zwischen einem Monolithischen Kernel und einem Minimalkernel ist der Microkernel. Hierbei sind die meisten Systemdienste zwar als Aktivitätsträger im User Mode realisiert, allerdings erfolgt die Kommunikation über den Microkernel. Das macht ein Client-Server System besonders geeignet für Verteilte Systeme, da ein Systemaufruf ohne dass die Anwendung sich darum kümmern muss über den Microkernel an einen anderen Rechner weitergeleitet werden kann. Auch die Fehlertoleranz ist bei einem solchen System relativ hoch, da ein “hängender” Server Prozess nicht das ganze System lahm legt, sondern einfach neu gestartet werden kann. Der grosse Nachteil dieser Architektur sind die vielen Context Switches, die das System nur mässig effizient machen.



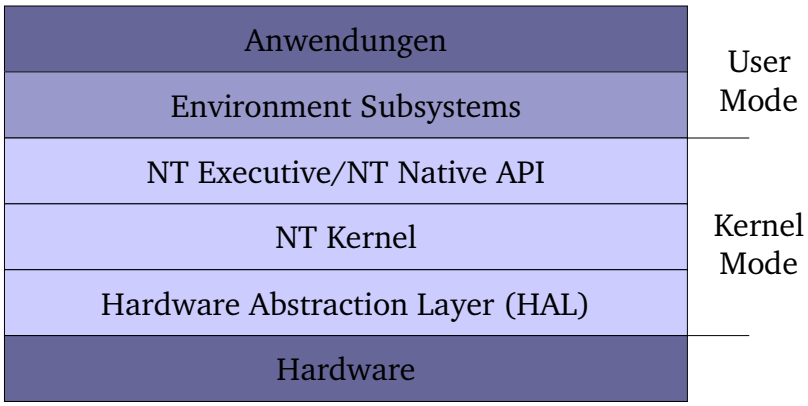
2) Umsetzung der Desing Goals in NT 5.0

Im diesem Teil soll es darum gehen zu zeigen, wie die Ideen und Konzepte aus dem letzten Abschnitt in Windows NT 5.0 (aka Windows 2000) umgesetzt wurden. Dazu wird erst einmal ein grober Überblick über die NT 5.0 Systemarchitektur gegeben und dann genauer darauf eingegangen, was in den User- und Kernel-Mode Komponenten so alles passiert.

2a) Architektur-Überblick

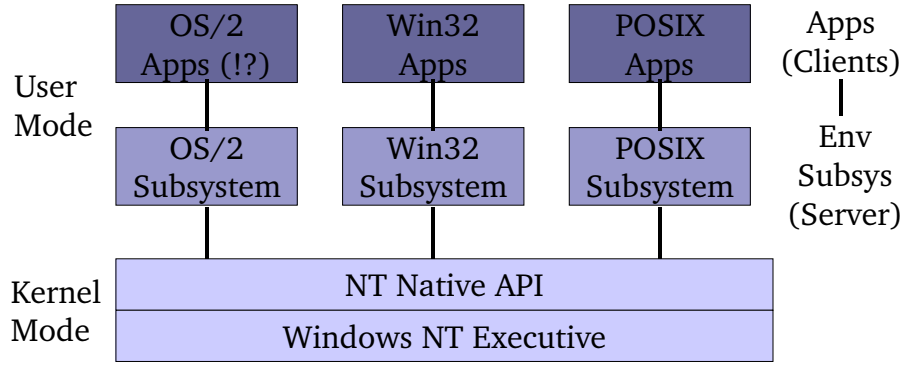
Bei Windows NT bediente man sich ähnlich wie bei Unix des Schichten-Modells um das System in mehr oder weniger überschaubare Teile zu gliedern. Man stellt fest, dass die NT Executive, die zahlreiche Systemdienste zur Verfügung stellt, im Kernel Modus (siehe 1c.1) läuft. Das heisst, dass man pro Systemaufruf nur einmal in den

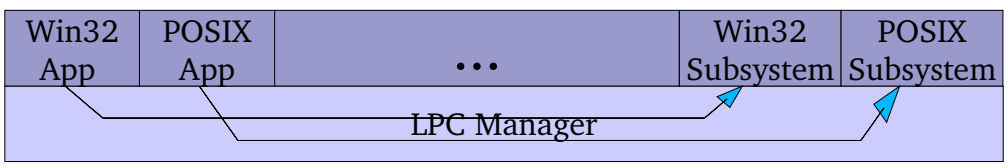
privilegierten Modus und zurück wechseln muss, was sich positiv auf die Geschwindigkeit des Systems auswirkt. Es fällt ausserdem auf, dass Anwendungen nicht direkt Systemaufrufe aus der Native API aufrufen, sondern indirekt über sogenannte Environment Subsystems, die verschiedene Betriebssystem-APIs z.B. von POSIX, OS/2, Win32, usw. zur Verfügung stellen, mit der NT Executive kommunizieren. Wie das genau funktioniert, werden wir uns im nächsten Abschnitt anschauen.



2b) Aufbau der User-Mode Komponenten

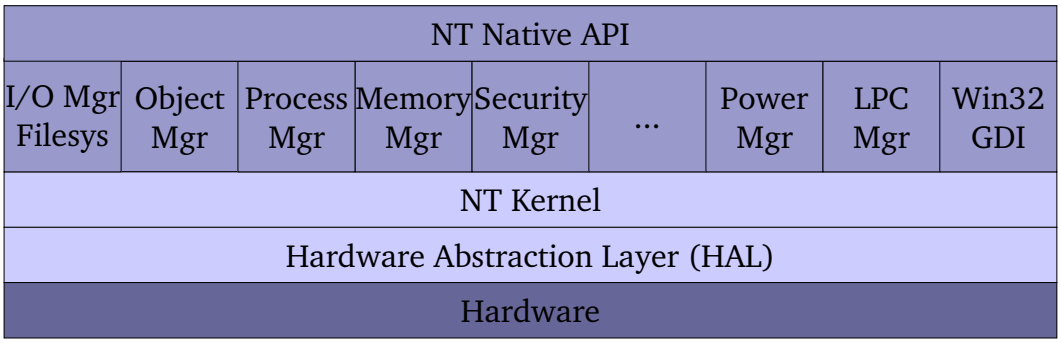
Da die NT Native API ständig durch neue Funktionen erweitert wird, Anwendungen aber trotzdem mit einem möglichst klar abgegrenzten, überschaubaren Satz an Systemaufrufen arbeiten sollten, macht es Sinn eine zusätzliche Schicht zwischen Anwendungen und Native API zu packen: die Environment Subsystems (ES). Zur Kommunikation zwischen Anwendungen und ESs wurde das Client-Server Modell gewählt, d.h. die Kommunikation findet über den LPC Service des Kernels statt (siehe 2c). Weitere Vorteile der Abstraktion der Native API sind die Möglichkeit, einerseits verschiedene BS-APIs zur Verfügung zu stellen und andererseits über Emulatoren DOS und Win16 Abwärtskompatibilität möglich zu machen.





2c) Aufbau des Kernels

Zunächst fällt beim Betrachten der Kernel Struktur auf, dass er eine Vielzahl von Modulen enthält. Allerdings sind diese Module nicht strikt voneinander getrennt, sondern zum Teil so stark miteinander verknüpft, dass das Diagramm eher eine Idealisierung als eine konkrete Darstellung der Wirklichkeit ist. Bei der Vielzahl der verschiedenen Dienste die im Kernel enthalten sind, erkennt man leicht, dass es sich um einen monolithischen Kernel (siehe 1c.1) handelt. Auch Module wie das GDI, die in früheren Versionen im User-Mode liefen, wurden aus Effizienz-Gründen in den Kernel verschoben ("Kernel" bezeichnet alles was im Kernel Mode läuft, "NT Kernel" ist der Teil des Kernels aus dem Diagramm).



Im Folgenden soll die Funktion und Bedeutung einiger wichtiger "Module" des Kernels erläutert werden.

- Object Manager (OM)
  Da alle Resourcen (Threads, Files, ..) unter Windows NT "Objekte" sind, spielt natürlich der OM, der alle Objekte verwaltet eine zentrale Rolle und ist eng mit fast jedem anderen Modul des Kernels verbunden. Objekte werden aus Sicherheitsgründen nach aussen nicht durch Zeiger auf Datenstrukturen repräsentiert, sondern durch Handles, die erst vom OM in Zeiger umgewandelt werden müssen. Damit Objekte wenn sie nicht mehr referenziert werden auch freigegeben werden, hat jedes Objekt einen Reference Counter.

- Security Manager (SM)
  Der SM vergibt Zugriffsrechte an Objekte, in dem jedem Objekt ein "Access Token" angehängt wird, der festlegt, wer in welcher Form auf ein Objekt zugreifen darf. Bei einem Zugriff auf ein Objekt konsultiert der OM erst den SM um die Rechte zu prüfen.

- Process Manager (PM)
  Der PM startet, stoppt, erstellt und löscht Prozess Objekte. Das Scheduling ist allerdings die Aufgabe des NT Kernels.

- Local Procedure Call Manager (LPCM)
  Die Kommunikation zwischen Anwendungen und Environment Subsystems wird, wie bereits erwähnt, über LPCs abgewickelt. Da der Nachrichtenaustausch sehr zeitkritisch ist, wird für diesen Mechanismus ein eigenes Kernel Modul zur Verfügung gestellt, das diese Aufgabe sehr effizient erledigt.

– NT Kernel
  Im NT Kernel findet u.A. das Thread Scheduling statt, wozu ein preemptiver Scheduler mit 32 Prioritätsstufen implementiert wurde. Auch die Context Switches zwischen User und Kernel mode werden vom NT Kernel durchgeführt. Der Kernel stellt ausserdem sogenannte “Dispatcher Objekte” zur Verfügung. Dabei handelt es sich um Mutexe, Events, Semaphoren, usw., die sehr eng mit dem Scheduler zusammen hängen.

– Hardware Abstraktion Layer (HAL)
  Die HAL abstrahiert die Hardware insofern, dass sie für z.B. Gerätetreiber eine einheitliche Schnittstelle in Form von Makros und Funktionen bietet die es ermöglicht, HW-unabhängig auf E/A Adressen, DMA Kanäle, usw. zuzugreifen. Bei der Windows Installation wird deswegen ein passendes HAL Image ausgewählt und in die Datei hal.dll kopiert, ohne die das System nicht lauffähig ist.

Quellen

[1] „Modern Operating Systems”, Second Edition
Andrew S. Tanenbaum, Prentice Hall, 2001

[2] „Sliding through Operating Systems”
Daniel Menasce, George Mason University, 1997
http://cs.gmu.edu/~menasce/osbook/

[3] „Exokernel: an operating system architecture for application-level resource management“
Dawson R. Engler, M. Frans Kaashoek et al, 1995