Universität Siegen
Fachbereich 12: Elektrotechnik und Informatik
Dozent: Dipl.-Inform. Benedikt Meurer
Hauptseminar Softwarearchitekturen

Softwarearchitekturen
Windows Vista und Windows Phone 7

Vorgelegt von Frank Urrigshardt
Studiengang: Medieninformatik
Sommersemester 2011

Inhaltsverzeichnis

1 Einleitung

Im Folgenden wird die Softwarearchitektur von Windows Vista und Windows Phone 7 vorgestellt. Dabei wird der Schwerpunkt auf Windows Vista liegen. Zum einen ist das Desktop-Betriebssystem wesentlich komplexer und zum anderen gibt es leider nur wenig verwertbares Material zur Architektur von Windows Phone 7.
Zunächst wird der Begriff der Softwarearchitektur näher beleuchtet. Anschließend gibt es einen kurzen Abriss über die Geschichte von Windows, um einige historisch bedingte Eigenheiten der Architektur von Windows besser verstehen zu können. Danach wird die Architektur zunächst aus der Sicht eines Programmierers für Anwendungssoftware dargestellt, um mit diesen Erkenntnissen letztendlich in den Kern des Betriebssystems blicken zu können. Abschließend wird kurz die Architektur von Windows Phone 7 vorgestellt, ohne dabei zu sehr ins Detail zu gehen.

2 Softwarearchitektur

Als Einstieg in die Thematik soll an dieser Stelle zunächst der Begriff der *Softwarearchitektur* erläutert und somit ein Rahmen für die Präsentation der Anwendungsbeispiele *Windows Vista* und *Windows Phone 7* geschaffen werden.
Der Begriff der klassischen Architektur wurde hier auf Software(-systeme) angewandt um zu verdeutlichen, dass es sich um ein bekanntes Phänomen handelt, welches nun auch in einer anderen Disziplin der Beachtung bedarf. Das Kernproblem von Architektur ist es, ein funktionierendes Ganzes aus unterschiedlichsten Einzelteilen zu erschaffen. Dabei geht es nicht nur um die vorausgehende Planung, sondern auch um die Beschreibung der einzelnen Komponenten und deren Zusammenspiel. Hervorzuheben ist, dass jedes Element eine wichtige Rolle spielt, ganz egal ob es das Fundament, eine tragende Wand oder das Dach ist.
Die vorangehende Beschreibung von Architektur lässt sich mit geringen Änderungen leicht auf Software übertragen. Denn auch in der Softwarearchitektur geht es um die Planung und Beschreibung einer Anwendung auf Basis ihrer einzelnen Komponenten. Die wichtigsten software-architektonischen Aspekte sind zum Einen die Software-Struktur(en) eines Systems, sowie dessen Software-Bausteine. Zum Anderen sind die Eigenschaften ebendieser Bausteine und deren Beziehungen untereinander von großem Interesse, um das komplexe 'Innenleben' einer Anwendung angemessen zu erfassen. Diese Bausteine können hierbei Schlüsselklassen, Schnittstellen, Komponenten, Frameworks, Subsysteme und Module sein.[1]

[1] Vgl. Vogel et al., S. 42-50

3 Windows Vista

In diesem Kapitel wird der Aufbau und die Funktionsweise von Windows Vista erläutert und veranschaulicht. Ich habe mich dazu entschieden, zunächst einen kurzen historischen Überblick über die Entwicklung von Windows zu geben. Dieser ermöglicht ein besseres Verständnis der grundlegenden Architektur, welche sich über viele Jahre bis hin zur aktuellen Version weiterentwickelt hat.

3.1 Historische Entwicklung

Die geschichtliche Entwicklung der Betriebssysteme von Microsoft lässt sich grob in drei große Abschnitte unterteilen: MS-DOS (**M**icro**S**oft **D**isc **O**perating **S**ystem), MS-DOS-basiertes Windows und NT-basiertes Windows.
Beim ab 1981 vermarkteten MS-DOS handelt(e) es sich um ein Einzelnutzer-Betriebssystem, das als einzige Benutzerschnittstelle lediglich über eine Kommandozeile verfügte (abgesehen von einer Batch-Schnittstelle). Die jüngste und wohl auch letzte Version ist *MS-DOS 8.0* aus dem Jahr 2000. Beim MS-DOS-basierten Windows wurde - wie der Name schon andeutet - lediglich eine grafische Benutzersschnitstelle auf das bereits bestehende 16-Bit-Betriebssystem MS-DOS aufgesetzt. Ebendieses steuerte nach wie vor die Maschine und verwaltete das Dateisystem. Da alle Programme im selben Adressraum des Hauptspeichers liefen war es nicht verwunderlich, dass ein Fehler in irgendeinem der Programme zu einem Komplettabsturz des Systems führte. Angefangen hat diese Produktreihe mit Windows 1.0, wobei erst die Version 3.0 überaus erfolgreich vermarktet werden konnte. Darauf folgende Betriebssysteme (*Windows 95/98/ME*) machten nun zwar einiges besser, sie ermöglichten beispielsweise die Nutzung von virtuellem Speicher, eine Prozessverwaltung sowie eine 32-Bit-Programmierschnittstelle (Win32-API) wurden eingeführt, aber in ihrem Kern lief noch immer MS-DOS und führte 16-Bit Assemblercode aus. Aufgrund dessen konnte die Instabilität dieser Produktreihe nie ganz ausgemerzt werden. Dieser Mißstand wurde bereits sehr früh von Microsoft entdeckt. So wurde parallel zum MS-DOS-basierten Windows relativ früh ein NT-basiertes (**N**ew **T**echnology) Windows entwickelt. Das erste Betriebssystem dieser neuen Kategorie nannte sich *Windows NT 3.1* - angelehnt an das damals vorherrschende *Windows 3.1*. Der Kern war nun ein komplett erneuerter mit 32-Bit Architektur, bei dem vor allem auf Sicherheit und Zuverlässigkeit geachtet wurde. Um größtmögliche Kompatibilität zwischen den verschiedenen Windows-Versionen zu gewährleisten, wurde auch bei NT weiterhin auf die Win32-Programmierschnittstelle gesetzt. Die Umstiegshürde sollte auf diese Weise möglichst umgangen werden, da alle bisherigen Anwendungen auf MS-DOS-basierten sowie auf

NT-basierten Windows-Versionen genutzt werden konnten.[2] (Siehe Abbildung 1)

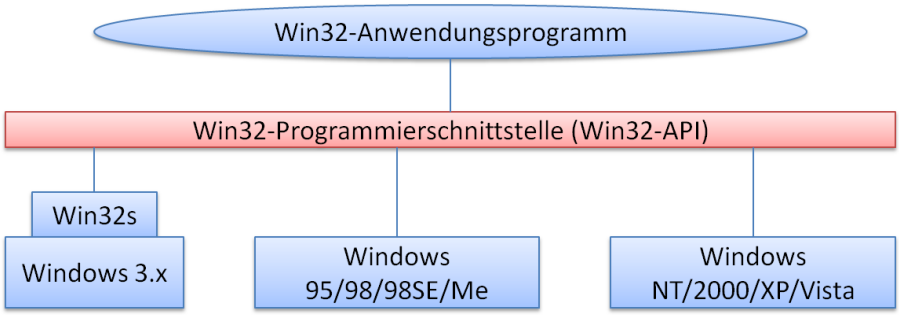

Abbildung 1: Die Win32-API

Das letzte MS-DOS-basierte Windows, *Windows ME*, wurde 2001 schließlich vom NT-basierten *Windows XP* komplett abgelöst und letzteres gewann das Vertrauen der Kunden. Heute aktuelle Betriebssysteme wie *Windows Vista* und vor allem *Windows 7* basieren ebenfalls auf dem NT-Kern und bisher ist noch kein Ende dieses Trends in Sicht.

3.2 Programmierung

Bevor die interne Architektur von Windows Vista detailliert erläutert wird, möchte ich zunächst einen etwas weniger tiefgreifenden Überblick über die unterschiedlichen Programmierschichten des Betriebssystems geben.
Die Struktur der Programmierschichten lässt sich auf Abbildung 2 reduzieren.

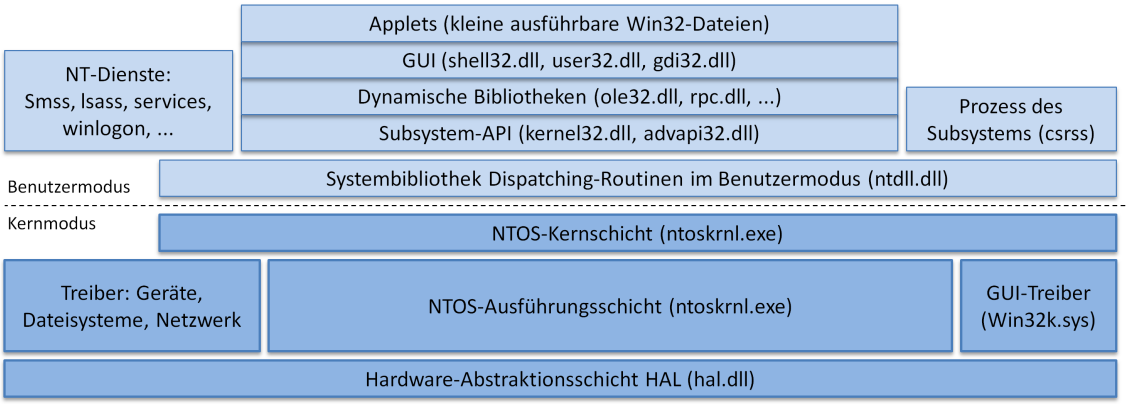

Abbildung 2: Programmierschichten in Windows

Das Kernmodusprogramm *ntoskrnl.exe* ist das Herzstück des NT-Betriebssystems (im Folgenden *NTOS* genannt). Es stellt die Schnittstellen für Systemaufrufe bereit, die vom Rest des Betriebssystems genutzt werden. Programmierer können nicht direkt Software

[2] Vgl. Tanenbaum, S. 936-941

entwickeln, welche direkt auf Funktionen der NTOS-Kernschicht zugreift. Sie müssen die Funktionen der APIs nutzen, welche als Subsysteme im Benutzermodus zur Verfügung stehen (oberhalb der NTOS-Schichten) und die Aufrufe in den Kern delegieren. Auch wenn nahezu alle Windows-Anwendungen und Windows-Code die Win32-API hierfür benutzen, gibt es dennoch unterstützte Alternativen. So verfügten alle NT-Betriebssysteme bis Windows 2000 noch über 3 sogenannte *Personalities*. Hierzu gehörten OS/2, POSIX und Win32. OS/2 wurde in Windows XP nicht mehr weiter verwendet, aber die POSIX-API kann noch immer unter Verwendung eines Microsoft Pakets (Services for UNIX) unter dem Namen *Interix* genutzt werden.
Eine Ausnahme hierzu stellt *.NET* dar. Hierbei handelt es sich nicht um ein Subsystem, welches direkt über den NTOS-Kernschichten liegt. Es baut vielmehr auf der Win32-Schnittstelle auf und fungiert in vielen Situationen lediglich als ein Wrapper um einzelne Win32-Funktionen. Allerdings sind die Schnittstellen vereinfacht worden und es wird eine größere Vielfalt an Objekttypen unterstützt. Ein weiterer wichtiger Vorteil ist, dass durch die .NET-Laufzeitumgebung eine automatische Speicherbereinigung zur Verfügung gestellt wird (analog zum *Java Garbage Collector*).[3]

3.2.1 NT-Programmierschnittstelle

Zusätzlich zu den eben vorgestellten Programmierschnittstellen Win32, .NET und POSIX gibt es natürlich noch eine native NT-Programmierschnittstelle. Die Funktionen dieser API, welche wie die anderen Schnittstellen auch im Benutzermodus zur Verfügung steht, werden in der NTOS-Ausführungsschicht implementiert. Es handelt sich hier also um die API, die den Systemaufrufen im Kernmodus am nächsten steht.
Wie bereits erwähnt, basieren die meisten Windows-Anwendungen auf der Win32-API oder .NET. Es stellt sich die Frage, warum nicht gleich die native NT-API benutzt wird. Der Hauptgrund hierfür ist, dass Microsoft keine allzu großen Details über diese Systemaufrufe veröffentlicht hat. Anwendungen auf Basis von der NT-API sollen vermieden werden, da diese nicht auf MS-DOS-basierten Betriebssystemen ausführbar wären. Wird mit Hilfe der gut dokumentierten Win32-API gearbeitet, so wird die Software von sämtlichen Microsoft-Betriebssystemen (mehr oder weniger gut) unterstützt.
Die Systemaufrufe der NTOS-Ausführungsschicht operieren wiederum auf *Kernmodusobjekten*. Dabei handelt es sich unter anderem um Dateien, Prozesse, Threads, Pipes und Semaphore. Diese Objekte lassen sich grob in 4 Typen unterteilen: Synchronisation, Ein-/Ausgabe, Programm und Win32-GUI. Mit Hilfe der NT-API können nun neue Kernmodusobjekte erzeugt und auf vorhandene zugegriffen werden. Jeder dieser Systemaufrufe, der ein neues Objekt erzeugt oder auf ein bestehendes zugreift, liefert als Ergebniss ein

[3] Vgl.Tanenbaum, S. 942-945

Handle zurück. Mit dessen Hilfe können letztendlich Operationen auf dem Kernmodusobjekt ausgeführt werden. Dabei ist zu beachten, dass Handles in der Regel zu dem Prozess gehören der sie erzeugt hat und auch nur in diesem Prozess verwendet werden können. Teilweise ist es jedoch möglich einen Handle zu duplizieren und in die Handle-Tabelle eines anderen Prozesses auf sichere Weise einzutragen.

```
NtCreateFile(FileHandle, FileNameDescriptor, Access, ...)
```

Abbildung 3: Beispiel für einen NT-API-Aufruf zum Erzeugen einer Datei.

Zudem verfügt jedes Kernmodusobjekt über einen sogenannten *Sicherheitsdeskriptor*. Dieser dient der Verwaltung von Rechten. So wird geregelt, wer auf ein Objekt zugreifen und welche Operationen ausgeführt werden dürfen. Bei oben erwähnter Duplizierung von Handles können auch Änderungen an den Rechten vorgenommen werden. So kann einem anderen Prozess z.B. ein modifizierter Handle übergeben werden, der nur Lesezugriffe erlaubt.
Die Kernmodusobjekte werden zentral vom *Objekt-Manager* verwaltet. Er erstellt auf bereits erwähnte Anfrage ein Handle zu einem Objekt, das anhand seines Namens identifiziert wird. Weiterhin bietet er eine einheitliche Verwaltung der Lebenszeit von Objekten sowie deren Sicherheitseinstellungen. Kernmodusobjekte, die hier verwaltet werden, werden natürlich nicht nur auf Anfrage von Anwendungen erzeugt, sondern auch das Betriebssystem selbst nutzt sie auf vielfältige Weise. Ein passendes Beispiel hierfür sind die an den PC angeschlossenen Geräte. Für jedes von ihnen wird ein repräsentatives *Geräteobjekt* erzeugt. Zur Steuerung des Geräts wird wiederum ein *Treiberobjekt* erstellt. Dieses bietet eine einheitliche Schnittstelle zur Steuerung eines Gerätetyps und 'übersetzt' die Anfragen in die Gerätespezifischen Anweisungen. Gerätetreiber werden hierbei als *permanent* markiert, d.h. sie bleiben so lange erhalten, bis sie explizit beendet werden oder das Betriebssystem heruntergefahren wurde - auch wenn kein Handle mehr darauf verweist. So müssen diese nicht für einzelne Prozesse neu erzeugt, sondern nur ein Handle vom Objekt-Manager angefordert werden.[4]

3.2.2 Win32-Programmierschnittstelle

Im Gegensatz zur NT-Programmierschnittstelle wurde die Win32-API von Microsoft öffentlich gemacht und ist zudem vollständig dokumentiert. Die Win32-Funktionsaufrufe sind implementiert als Bibliotheksfunktionen, die zum großen Teil - wie bereits im Vorfeld erwähnt - lediglich als Wrapper für NT-Systemaufrufe fungieren. Ein Beispiel für einen Wrapper wäre der Win32-Aufruf `CreateProcess`, der wiederum `NtCreateProcess`

[4] Tanenbaum, S. 945-949

der NT-API aufruft und Win32-Parameter (wie z.B. Pfadnamen) daran anpasst. Zudem gibt es jedoch auch Funktionen, welche direkt im Benutzermodus arbeiten und keine Delegation des Aufrufs in darunterliegende Ebenen erfordern. Fast alle Funktionen der NT-Schnittstelle können so über die Win32-Schnittstelle genutzt werden. Letztere wird nur wenig im Laufe der Weiterentwicklung von Windows verändert: Es werden hauptsächlich neue Funktionen hinzugefügt, weniger jedoch bereits bestehende modifiziert.
Man sollte also meinen, dass alle Anwendungen die auf älteren Windows-Versionen liefen auch auf anderen, aktuelleren Versionen problemlos auszuführen sein sollten. Diese theoretische Sicht lässt sich jedoch leider nicht in die Praxis übertragen. Das Betriebssystem ist schlicht derart komplex, dass selbst unscheinbarste Änderungen zu Inkompatibilität führen können. In vielen Fällen sind aber auch die Programmierer von Anwendungssoftware selbst schuld daran, da sie in ihrem Programmcode Überprüfungen nach spezifischen Windows-Versionen benutzen.
Nun ein etwas genauerer Blick auf die Funktionalitäten, die durch die Win32-API bereitgestellt werden. Dazu gehören die Erzeugung und Verwaltung von Prozessen und Threads. Die Kommunikation zwischen Prozessen und Threads wird durch Aufrufe ermöglicht, wie z.B. das Erzeugen, Zerstören und Benutzen von u.a. Mutexen, Semaphoren und Ereignissen.
Zudem werden *Memory-Mapped-Dateien* und *Demand Paging* ermöglicht. Ersteres bedeutet, dass die Speicherverwaltung einem Prozess ermöglicht, eine Datei in seinen virtuellen Speicherbereich einzublenden. Threads innerhalb des Prozesses können somit aus der Datei lesen und in sie hinein schreiben, ohne auf die Festplatte zugreifen zu müssen. Es handelt sich also um eine Kopie, die im Hauptspeicher vorgehalten wird. Durch Demand Paging werden Änderungen an der Kopie je nach Bedarf auf das Original auf der Festplatte übertragen und umgekehrt.
Desweiteren bietet Win32 über 60 Aufrufe für Dateiein- und -ausgabe. Dazu gehören das Erstellen, Bearbeiten und Löschen von Dateien und Verzeichnissen, aber auch Setzen und Ändern von Dateiattributen. Grundlegend sind Dateien aus der Sicht von Win32 jedoch nichts weiter als lineare Bytefolgen. Durch NTFS wird zudem ermöglicht, Dateien zu verschlüsseln. Es können bei Bedarf sogar ganze Volumes verschlüsselt werden (Anmerkung: Seit Windows 7 ist es sogar möglich, die Partition auf der das Betriebssystem installiert ist komplett zu verschlüsseln). Durch diese Maßnahme können die Dateien beispielsweise nicht mehr durch Nutzung eines Linux Betriebssystems ausgespäht werden. Nur durch ein gültiges Zertifikat lassen sich die Daten unter Windows rekonstruieren.
Die soeben beschriebene Dateiein- und -ausgabe wird durch die maschinennahen Schichten von Windows grundlegend asynchron ausgeführt. Wird ein Schreibzugriff auf die Festplatte veranlasst, so wird diese 'im Hintergrund' ausgeführt und der aufrufende Thread

kann weiter seine Arbeit verrichten. Dies ist allerdings kein per se sicheres Konzept für die Ein-/Ausgabe. Daher werden Mechanismus bereitgestellt, die paralleles Schreiben in eine Datei verhindern.
Bisher wurden nur maschinennahe Systemschnittstellen vorgestellt, jedoch werden auch viele GUI-Operationen von der Win32-API angeboten. Dazu gehören Aufrufe zum Erzeugen, Manipulieren, Schließen, etc. von Fenstern, bis hin zu Scrollbalken und Icons. Aber auch Funktionen zum Zeichnen und Füllen von geometrischen Objekten stehen zur Verfügung, sowie eine Verwaltung von Farbpaletten. Nicht zuletzt gibt es ebenfalls Aufrufe für den Umgang mit Ein- und Ausgabegeräten wie z.B. Maus, Joystick, Tastatur, Drucker, etc. Die GUI-Operationen arbeiten allerdings nicht auf darunterliegenden NT-Systemaufrufen, sondern mit dem *win32k.sys*-Treiber. Dabei werden nur Bibliotheken des Benutzermodus genutzt.[5]

3.3 Systemarchitektur

Nachdem nun die Programmierschichten vorgestellt wurden, die vor allem für Programmierer von Code für den Benutzermodus interessant sind, soll nun ein näherer Blick auf die interne Organisation geworfen werden. Dabei geht es nicht nur um die Struktur der Komponenten des Betriebssystems, sondern auch darum was sie im Einzelnen leisten, wie sie untereinander zusammenarbeiten und letztendlich mit Benutzerprogrammen interagieren.

3.3.1 Betriebssystemstruktur

Im vorigen Kapitel wurde bereits die Schichtenarchitektur von Windows Vista anhand von Abbildung 2 vorgestellt. Der Fokus lag dort eher auf der detaillierteren Darstellung des Benutzermodus und der Programmierschnittstellen. In diesem Kapitel wird der Fokus nun auf die unteren Schichten gelegt, die im Kernmodus laufen. Sie sind zwangsläufig maschinennäher und bilden das Rückrad des Betriebssystems. (Siehe Abbildung 4)
Der NTOS-Kern stellt die beiden zentralen Schichten dar. Eine kleinere NTOS-Kernschicht (*kernel*) setzt die CPU-Verwaltung um: Sie implementiert Thread-Scheduling und Synchronisation, sowie diverse Interrupts und Unterbrechungsroutinen. Hierbei mag es durchaus zu Verwirrung führen, dass ein Teil des NTOS-Kerns wiederum als Kern bezeichnet wird. Die darunter liegende größere Schicht wird Ausführungsschicht genannt (*executive*). Sie beinhaltet die meisten Dienste. Die Darstellung der NTOS-Kernschicht als übergeordnete Schicht ist dadurch zu erklären, dass sie genau die Mechanismen für Unterbrechungen und Interrupts implementiert, die zum Übergang vom Benutzer- in den Kernmo-

[5] Tanenbaum, S. 949-953

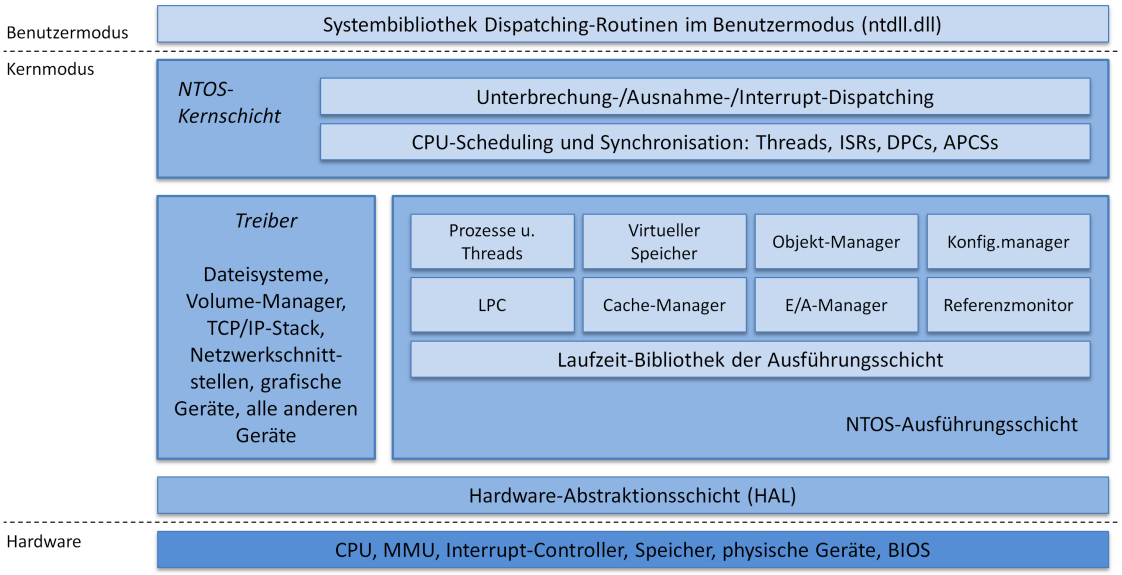

Abbildung 4: Der Windows-Kernmodus

dus verwendet werden.

Die unterste Schicht vor der Hardware ist die *Hardware-Abstraktionsschicht*. Dieser ***H**ardware **A**bstraction **L**ayer* (kurz: HAL) dient dazu, die maschinennahen Hardwaredetails zu abstrahieren und somit eine einheitliche Schnittstelle für den Zugriff auf diverse Hardware zur Verfügung zu stellen.

Die letzte Komponente des Kernmodus sind die Gerätetreiber. Alle Kernmodusfunktionen, die nicht Teil des HAL oder des NTOS-Kerns sind, werden als *Gerätetreiber* implementiert. Neben der bekannten Nutzung von Treibern für physische Geräte, wie z.B. DVD-Laufwerke oder Monitore, werden diese u.a. auch für Dateisysteme, sowie Kernerweiterungen in Form von Antivirensoftware oder Programmen zum digitalen Rechtemanagement (DRM) benutzt.[6]

3.3.2 Hardware-Abstraktionsschicht

Da die Hardware-Abstraktionsschicht einige wichtige Funktionen erfüllt, möchte ich an dieser Stelle näher darauf eingehen. Eine Tatsache ist, dass in der Welt der Computer unterschiedlichste Hardwareplattformen zu finden sind. Die maschinennahen Schichten eines Betriebssystems sehen sich mit variierenden Hardwarefunktionen, wie beispielsweise Geräteregistern, Interrupts und DMA konfrontiert. Ein wichtiges Ziel bei der Programmierung von Windows war stets, maximale Portabilität für unterschiedlichste Hardwareplattformen zu ermöglichen. Der Idealfall wäre natürlich, wenn das Betriebsystem mit einem Compiler so übersetzt werden könnte, dass es auf jeder Hardwareplattform oh-

[6] Tanenbaum, S. 957-958

ne manuelle Anpassung funktionsfähig wäre. Alle nicht von der Hardware abhängigen Komponenten sind in dieser Form portabel, da sie mit internen Datenstrukturen und Abstraktionen arbeiten. Bei den abhängigen Komponenten ist dies nicht so leicht machbar.
Ein Beispiel für einen Fall, bei dem die Hardwarespezifika nicht hinter einem Compiler versteckt werden können, wäre der Unterschied zwischen einem x86- und einem SPARC-System. Hier sind nicht einfach nur die Befehlssätze der Prozessoren unterschiedlich, sondern die gesamten Prozessorarchitekturen sind stark verschieden. Hinzu kommt, dass zwar nahezu 99% des NTOS-Kerns in C geschrieben sind, der restliche Anteil jedoch in Assembler. Dieser Anteil muss zwangsläufig manuell portiert werden.
Abgesehen von diesen großen Problemen, gibt es ebenfalls unzählige kleinere Stolpersteine die eine Portierung erschweren. Selbst unterschiedliche Hauptplatinen für die selbe Architektur können in der Realisierung stark variieren - ebenso die Implementierung von Synchronisationsoperationen (wie beispielsweise Spinlocks) bei diversen CPU-Versionen.
Um diese Abhängikeiten des Kerns nun möglichst zu minimieren, wurde von Microsoft der Versuch unternommen, diese durch die Einführung einer dünnen Schicht vor dem Rest des Kerns zu verstecken und diesen somit unabhängig von Hardwarespezifika zu machen. Der *HAL* bietet somit einheitliche Schnittstellen, auf denen der Kern ohne genauere Kenntnisse über die Hardware arbeiten kann. Dieser Tatsache ist es zu verdanken, dass der Kern und Treiber in wesentlich geringerem Umfang geändert werden müssen (falls überhaupt), um das Betriebssystem auf einen neuen Prozessor zu portieren. Zudem ist die Portierung des HAL wesentlich leichter, da hier alle Hardwareabhängigkeiten zentral zusammenlaufen und nicht über den ganzen Kern verstreut sind. Eigens für diesen Zweck wird von Microsoft das HAL Development Kit zur Verfügung gestellt. So kann Windows mit abschätzbaren Arbeitsaufwand für nicht der Norm entsprechender Hardware portiert werden.
Ein weiterer wissenswerter Aspekt - ohne jedoch zu sehr ins Detail gehen zu wollen - ist, dass es ebenfalls der HAL ist, der beim Starten des Betriebssystems eine Kommunikation mit dem BIOS aufbaut. Die Systemkonfiguration (angeschlossene Hardware, etc.) wird ausgelesen und anschließend vom HAL in die Registrierung geschrieben.[7]

3.3.3 Kernschicht

In diesem Kapitel soll die Kernschicht im Sinne der 'oberen' Schicht des NTOS im Groben vorgestellt werden (siehe Abbildung 4). Diese wird auch *Kernel Layer* genannt. Wie andere Schichten auch, dient auch sie dazu, darunter liegende Funktionalitäten zu abstrahieren und eine einheitliche Schnittstelle zu deren Nutzung zur Verfügung zu stellen. In

[7] Tanenbaum, S. 958-961

diesem Fall sind es zwei Hauptaufgaben die erfüllt werden: Zum einen die Verwaltung der CPU und zum anderen eine maschinennahe Unterstützung für zwei Synchronisationsmechanismen, die an dieser Stelle jedoch nicht weiter erläutert werden sollen.
Zur Verwaltung der CPU stellt die Kernschicht *Threads* für höhere Schichten bereit. Es werden aber auch Ausnahmebehandlungen und diverse Arten von Interrupts implementiert. Die Datenstrukturen, die letztendlich genutzt werden um einen Thread zu realisieren, werden von der darunter liegenden Ausführungsschicht implementiert. Wann welcher Thread welche CPU beanspruchen darf und wie Threads erfahren ob ein anderer bereits seine Arbeit verrichtet hat, wird ebenfalls in der Kernschicht umgesetzt. Man spricht hierbei auch von *Scheduling* und *Synchronisation*.[8]

3.3.4 Ausführungsschicht

Die Ausführungsschicht stellt nun die 'untere' Schicht des NTOS dar. Sie wird auch *Executive Layer* genannt und wurde in C programmiert. Wie zuvor erwähnt ist sie dank des HAL weitestgehend unabhängig von der zugrunde liegenden Hardware und konnte somit leicht auf andere Prozessoren portiert werden. Dazu gehören beispielsweise MIPS, x86, PowerPC, x64, etc. Einzig die Speicherverwaltung benötigte manuelle Anpassung, da diese nicht komplett unabhängig von der Hardware realisiert werden konnte.
Die Komponenten der Ausführungsschicht wurden nach dem Prinzip der Datenkapselung implementiert. So werden interne Datenstrukturen und Funktionen vor anderen Komponenten verborgen und externe Schnittstellen angeboten, die von allen Komponenten der Ausführungsschicht nutzbar sind. Zudem werden einige von ihnen über die *ntoskrnl.exe* den Treibern zur Verfügung gestellt. Wie in Abbildung 4 zu sehen ist, werden die meisten Komponenten als 'Manager' bezeichnet. Damit soll verdeutlicht werden, dass z.B. Speicher oder Prozesse an dieser Stelle verwaltet werden.
Im Folgenden wird nun eine Teilmenge der Komponenten der Ausführungsschicht kurz vorgestellt. Die Kernmodusobjekte der Ausführungsschicht, wie unter anderem Dateien, Threads, Prozesse, Treiber und Semaphore, haben einige grundlegende Gemeinsamkeiten. Es ist daher durchaus von Nutzen, diese Objekte zentral zu verwalten. Diese Aufgabe übernimmt der *Objekt-Manager*. Im Detail kümmert er sich ebenfalls um die Bereitstellung von Speicherplatz für neue Objekte und die Speicherfreigabe beim Löschen eines Objekts. Zudem wurde im vorigen Kapitel darauf verwiesen, dass höhere Schichten sogenannte Handles benötigen, um mit den Kernmodusobjekten interagieren zu können (ähnlich Pointern in C++). Der Objekt-Manager stellt diese zur Verfügung und speichert zudem die Anzahl der derzeit ausgegebenen Handles sowie die Anzahl der existierenden Kernmoduszeigerreferenzen. Sind keine Verweise auf das Objekt mehr vorhanden, kann

[8] Tanenbaum, S. 962-963

es je nach Einstellung der Lebensdauer gelöscht und der Speicher freigegeben werden.
Wie der Name bereits andeutet, ist der *E/A-Manager* für die Verwaltung von Ein- und Ausgabegeräten verantwortlich. Zudem stellt er Dienste bereit die eine Konfiguration von Geräten ermöglicht, sowie den Zugriff auf ebendiese. Von besonderem Interesse beim E/A-Manager ist, dass er nicht nur physische Geräte einbindet. So werden beispielsweise Dateisysteme und Netzwerkstacks ebenfalls dynamisch vom Kern geladen. Zusammen mit der immer besser unterstützten Möglichkeit, dass viele Gerätetreiber nun im Benutzermodus ausgeführt werden können, ist dies ein enormer Vorteil. Bei früheren Windows-Versionen trat relativ häufig der bekannte *Blue Screen Of Death* auf, dessen Hauptursache Fehler in Gerätetreibern waren, die im Kernmodus ausgeführt wurden. Für ein stabiles Betriebssystem ist ein schlanker Kern somit vorteilhaft.
Nahezu selbsterklärend ist der *Prozess-Manager*, der sich um das Erzeugen und Beenden von Prozessen und Threads kümmert. Anzumerken ist hier als Erinnerung, dass er nicht für Scheduling oder Synchronisation verantwortlich ist. Für diesen Aspekt ist die Kernschicht zuständig. Er verfügt außerdem über eine Handle-Tabelle, die Prozesse nutzen können um mit Kernmodusobjekten zu interagieren.
Zuletzt soll an dieser Stelle noch der *Konfigurationsmanager* vorgestellt werden, der die Registrierung implementiert. Wie bereits besprochen wird während des Bootvorgangs vom HAL die Systemkonfiguration aus dem BIOS ausgelesen und in die Registrierung geschrieben. Die dadurch erfasste Daten werden vom Konfigurationsmanager letztendlich im Dateisystem in sogenannten Hives gespeichert. Beim Startvorgang wird das wichtige SYSTEM-Hive zunächst in den Speicher geschrieben und erst wenn die Ausführungsschicht alle Komponenten initialisiert hat, wird die Konfiguration ins Dateisystem übertragen.[9]

4 Windows Phone 7

Bei Windows Phone 7 handelt es sich um die Weiterentwicklung der *Windows Mobile* Reihe, welche wiederum auf *Windows CE* für eingebettete Systeme, Thin Clients und Handhelds basiert.[10] Die wohl auffälligste Änderung ist, dass die UI komplett überarbeitet wurde und nun nicht mehr an die Desktop-Version von Windows erinnert. Sie ist speziell für den Gebrauch auf mobilen Endgeräten per Multi-Touch-Bedienung entwickelt worden. Doch es gibt noch weitere einschneidende Veränderungen: Bei Windows Phone 7 handelt es sich nun um ein geschlossenes System, ähnliches Apples iOS Betriebssystem. So ist kein direkter Zugriff auf das Dateisystem mehr möglich. Neue Anwendungen können lediglich in Form von sogenannten Apps vom Microsoft *Windows Phone Mar-*

[9] Tanenbaum, S. 967-971
[10] Wikipedia: Microsoft Windows CE — Wikipedia, Die freie Enzyklopädie

ketplace erworben werden. Datenaustausch mit dem PC ist nur über Synchronisation mit Hilfe von *Zune* möglich, sehr ähnlich zu Apples *iTunes*. Um nicht mit stark variierender bzw. zu schwacher Hardware kämpfen zu müssen, schreibt Microsoft zudem Mindestanforderungen für Handys vor, die Windows Phone 7 als Betriebssystem nutzen wollen.[11]

4.1 Architektur

Die Architektur von Windows Phone 7 ähnelt in grobem Zügen der Architektur des großen Bruders für den Desktop-PC. Direkt über der Hardware sitzt der Kernel des Betriebssystems als unterste Schicht, der eine Abstraktion der Hardware vornimmt und für die Verwaltung von Speicher zuständig ist. Zudem werden im Kernel einige Sicherheitsrelevante Konzepte, sowie Netzwerkfunktionalitäten umgesetzt. Die nächsthöhere Schicht besteht aus drei großen Komponenten: App Model, Ui Model, Cloud Integration. Diese werden im weiteren Verlauf noch genauer erläutert. Diese beiden Schichten bilden den Kern des Betriebssystems. Darauf aufbauend läuft Microsofts .NET Laufzeitumgebung *Common Language Runtime* (CLR). Anwendungen für Windows Phone 7 werden demnach in der Sprache C# geschrieben. Microsoft bietet hierfür zwei Frameworks an, *XNA* und *Silverlight*. Zusätzlich können Inhalte in *HTML/Javascript* per integriertem Browser dargestellt werden.[12]

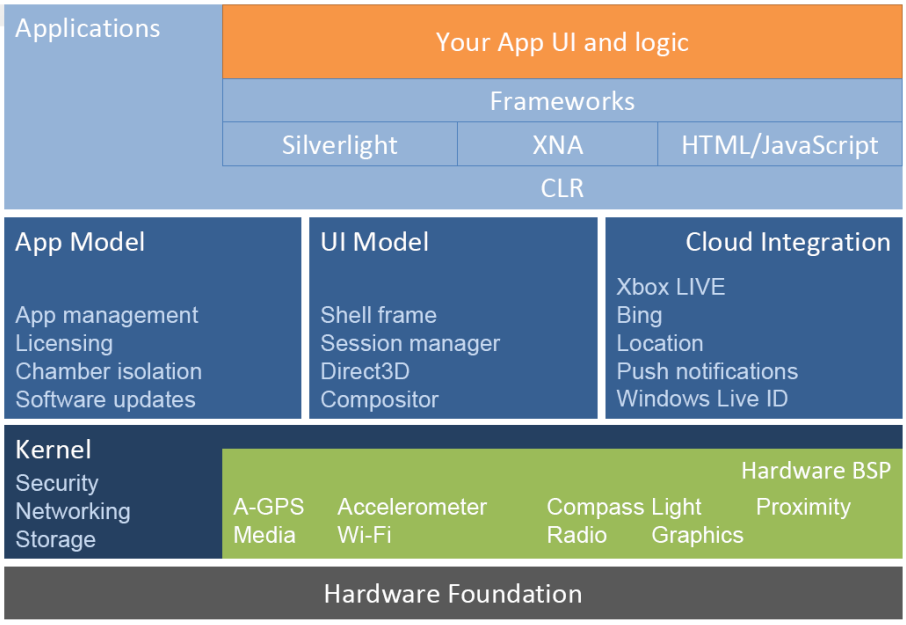

Abbildung 5: Die Windows Phone 7 Architektur

Bei XNA handelt es sich um ein Framework zur plattformübergreifenden Spieleprogram-

[11] Wikipedia: Windows Phone 7 — Wikipedia, Die freie Enzyklopädie
[12] Klug

mierung für die Plattformen Windows, *Xbox 360* und Windows Phone 7. Zur Darstellung von 2D und 3D Szenen wird *Direct3D* aus *DirectX 9.0c* verwendet.[13] Im Gegensatz zu XNA wird *Silverlight* für herkömmliche Anwendungen genutzt. Zunächst wurde es für sogenannte *Rich Internet Applications* entwickelt, welche im Browser mit einem Plugin wiedergegeben werden können. Programmierer, die zuvor Rich Internet Applications mit Silverlight erstellt haben, können ohne große Einstiegshürde Applikationen mit bekannten Werkzeugen erstellen. Dabei wird die Anwendungsoberfläche in der XML-Sprache *XAML* (eXtensible Application Markup Language) gestaltet. Die darunterliegende Applikationslogik wird in C# implementiert.[14]
Das *App Model* kümmert sich um die Verwaltung von Apps, deren Lizensierung, sowie um Softwareupdates und stellt zudem einen Sandbox-Mechanismus bereit. Eine einzelne App besteht lediglich aus einem einzigen Paket aus Metadaten, Icons, DLLs, etc. im XAP Format. Das App Model startet nur Apps, die eine gültige Marektplace Lizenz vorweisen können. Ausgeführt und installiert werden diese in einer Sandbox, einem speziellen Sicherheitsaccount mit den geringstmöglichen Berechtigungen. Mit dieser Maßnahme sollen Übergriffe von Schadsoftware auf das Mobiltelefon verhindert werden.
Das *UI Model* stellt grundlegende Funktionen zur visuellen Darstellung bereit, aber verwaltet auch den zeitlichen Verlauf von Seitenaufrufen. Das Konzept dahinter ist relativ einfach: Es gibt Applikationen, Seiten und Sessions. Applikationen bestehen hierbei aus einer Menge von Seiten. Eine Session wiederum speichert den Verlauf von durch den Anwender geöffneten Seiten - Applikationsübergreifend. So kann immer wieder zu einer früheren Seite zurückgesprungen werden. Das UI besteht insgesamt aus übereinanderliegenden Schichten, die vom *Shell Frame* organisiert und zur Darstellung zusammengesetzt werden. Wird eine Applikation gestartet, so wird der Startbildschirm also nicht vom Stack entfernt, sondern nur von der neuen Applikation überlagert.
Die *Cloud Integration* bietet darüber hinaus bekannte APIs zur Kommunikation mit bestehenden Web 2.0 Services (z.B. Windows Live, Xbox Live, bing), sowie die Möglichkeit eigene Services in seinen Anwendungen zu nutzen.[15]

Literatur

1. **Klug, Brian:** Windows Phone 7: The AnandTech Guide. 2011, [Online; Stand 6. Juni 2011] ⟨URL: `http://www.anandtech.com/show/2969/windows-phone-7-series-the-anandtech-guide/2`⟩

[13] WIKIPEDIA: XNA (Microsoft) — Wikipedia, Die freie Enzyklopädie
[14] WIKIPEDIA: Microsoft Silverlight — Wikipedia, Die freie Enzyklopädie
[15] PRENGEL

2. **Prengel, Frank:** Architektur der Anwendungsplattform von Windows Phone 7. 2011, [Online; Stand 6. Juni 2011] ⟨URL: http://www.microsoft.com/germany/msdn/webcasts/library.aspx?id=1032453977⟩

3. **Tanenbaum, Andrew S.:** Moderne Betriebssysteme. Pearson Studium, 2009, Pearson Studium, ISBN 9783827373427

4. **Vogel, O. et al.:** Software-Architektur: Grundlagen - Konzepte - Praxis. Spektrum Akademischer Verlag, 2008, ISBN 9783827419330

5. **Wikipedia:** Microsoft Silverlight — Wikipedia, Die freie Enzyklopädie. 2011, [Online; Stand 6. Juni 2011] ⟨URL: http://de.wikipedia.org/w/index.php?title=Microsoft_Silverlight&oldid=89707516⟩

6. **Wikipedia:** Microsoft Windows CE — Wikipedia, Die freie Enzyklopädie. 2011, [Online; Stand 6. Juni 2011] ⟨URL: http://de.wikipedia.org/w/index.php?title=Microsoft_Windows_CE&oldid=89260488⟩

7. **Wikipedia:** Windows Phone 7 — Wikipedia, Die freie Enzyklopädie. 2011, [Online; Stand 6. Juni 2011] ⟨URL: http://de.wikipedia.org/w/index.php?title=Windows_Phone_7&oldid=89636646⟩

8. **Wikipedia:** XNA (Microsoft) — Wikipedia, Die freie Enzyklopädie. 2011, [Online; Stand 6. Juni 2011] ⟨URL: http://de.wikipedia.org/w/index.php?title=XNA_(Microsoft)&oldid=88475963⟩