Betriebssysteme

Prof. Neeraj Suri, Dr. Stefan Winter

Wintersemester 2018-2019

Inhaltsverzeichnis

1 Geschichte

Am Anfang gab es noch keine Betriebssysteme, alle Programme liefen direkt auf der Hardware (*Bare Metal*). Da jedes Programm in irgendeiner Weise Daten verarbeiten muss, musste man für jedes Programm gewisse Eingaben zugänglich machen, und gewisse Ausgaben in einer für dem Benutzer nutzbaren Art ausgeben. Also musste man Code schreiben, der eigentlich garnichts mit dem Programm zu tun hat, sondern nur dazu dient, diese Eingaben und Ausgaben zu ermöglichen.

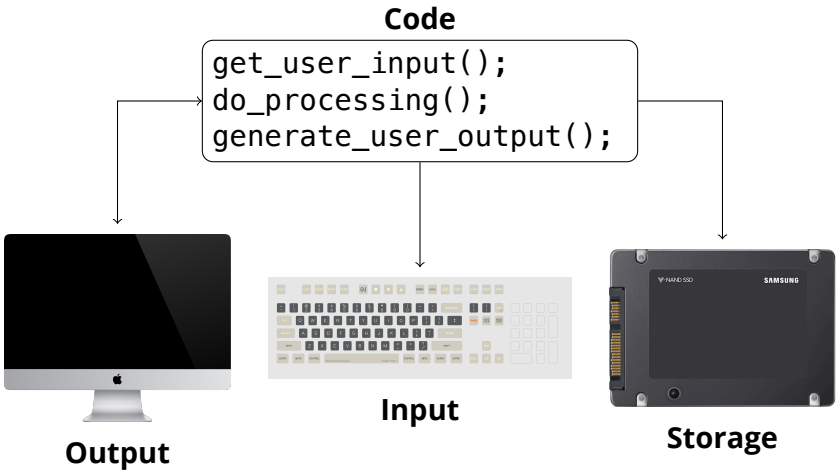


Abbildung 1: Programmierung auf Hardware ohne Abstraktion

1.1 Library

Irgendwann erkannte man dann, dass man sehr oft gewisse Funktionalität neu implementieren musste, weil unterschiedliche Programme ähnliche Eingaben erwarteten oder ähnliche Ausgaben erzeugten, und man diesen Code dafür widerholte. Eine Idee hier ist, den Code, den man häufig verwendet, in eine *Library* packt. Damit können diese Routinen wiederverdendet werden. Außerdem können Programme sich so mehr auf ihre eigentlichen Aufgaben konzentrieren.

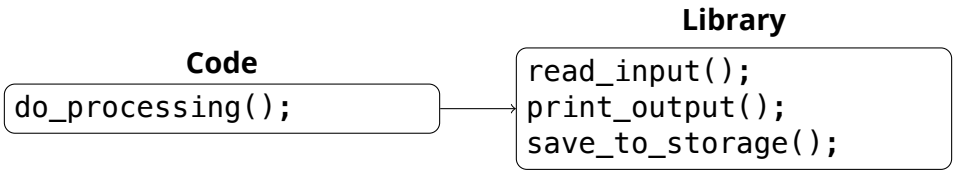


Abbildung 2: Ausbau von Funktionen in eine Library

Der Nachteil von diesem Ansatz war, dass diese Libraries immer wieder umge-

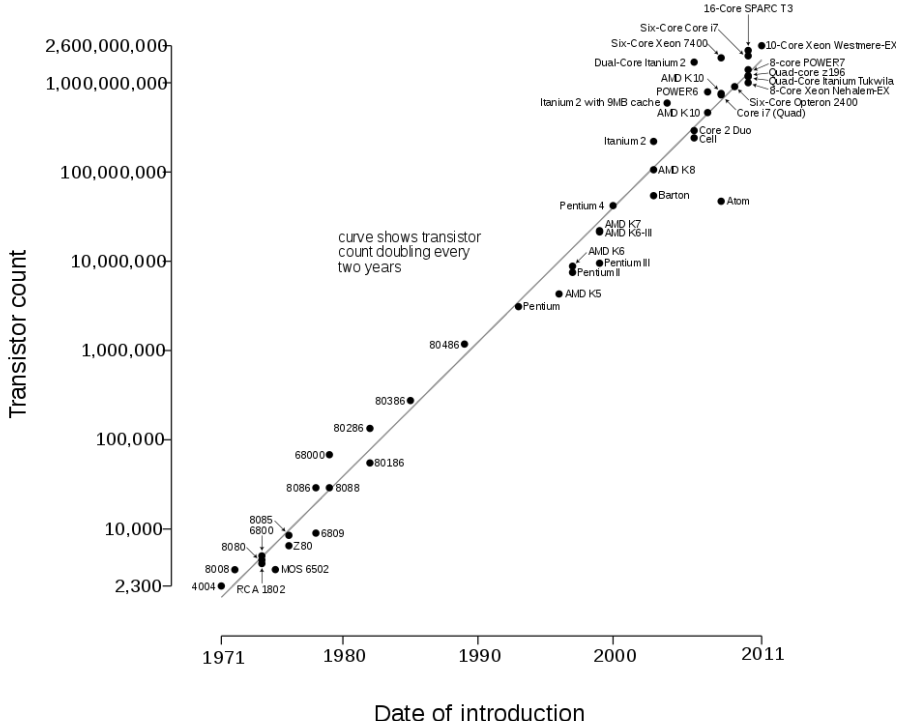


Abbildung 3: Hardwarekomplexitätswachstum seit 1971.

schrieben mussten, wenn ein anderer Computer verwendet wurde.

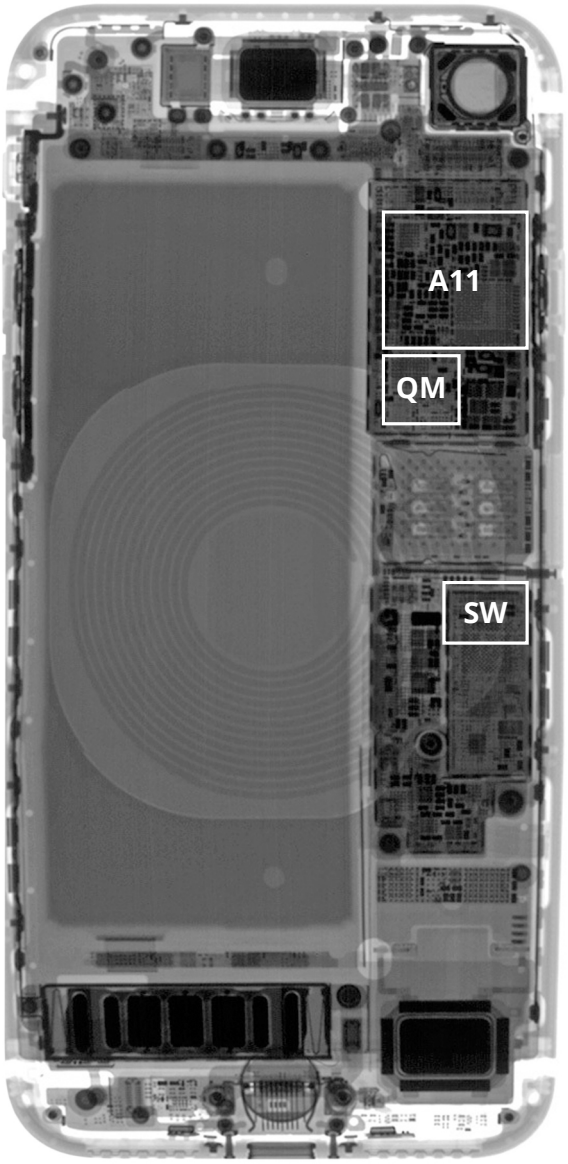
1.2 Exponentielles Wachstum von Hardwarekomplexität

Es ergab sich also die Schwierigkeit, dass die Hardware sich so rasant entwickelt. Dies ist in Abbildung 3 graphisch aufgezeigt, wobei man anmerken muss, dass die Skala logarithmisch ist. Das scheinbar lineare Wachstum der Transistoranzahl ist also tatsächlich ein exponentielles. Das nennt man auch Moore's law, nach dem Begründer von Intel.

1.3 Heute

Heutzutage sind Betriebssysteme *überall*. Von Fernsehehern zu Autos, Servern, Handys bis hin zu Operationssälen. Interessanterweise sind teilweise mehrere Betriebssysteme auf einem Gerät zu finden. Abbildungen 4 und 5 zeigen, dass sogar in den Geräten, die wir tagtäglich verwenden, oft mehr steckt, als wir denken.

Die Betriebssysteme, die wir benutzen, haben oft sehr viele Features eingebaut.



Apple A11 SoC
Betriebssystem: iOS (Darwin), UNIX-Derivat, POSIX kompatibel.

Apple Secure Enclave
Im A11-Chip eingebaut
Betriebssystem: Basiert auf dem L4 Mikrokernel

Qualcomm Snapdragon X16
LTE Modem
Betriebssystem: Unbekanntes RTOS, eventuell REX OS

Skyworks SkyOne SKY78140
CDMA Modem
Betriebssystem: Unbekannt

Abbildung 4: Betriebssyteme auf Konsumergeräten am Beispiel iPhone

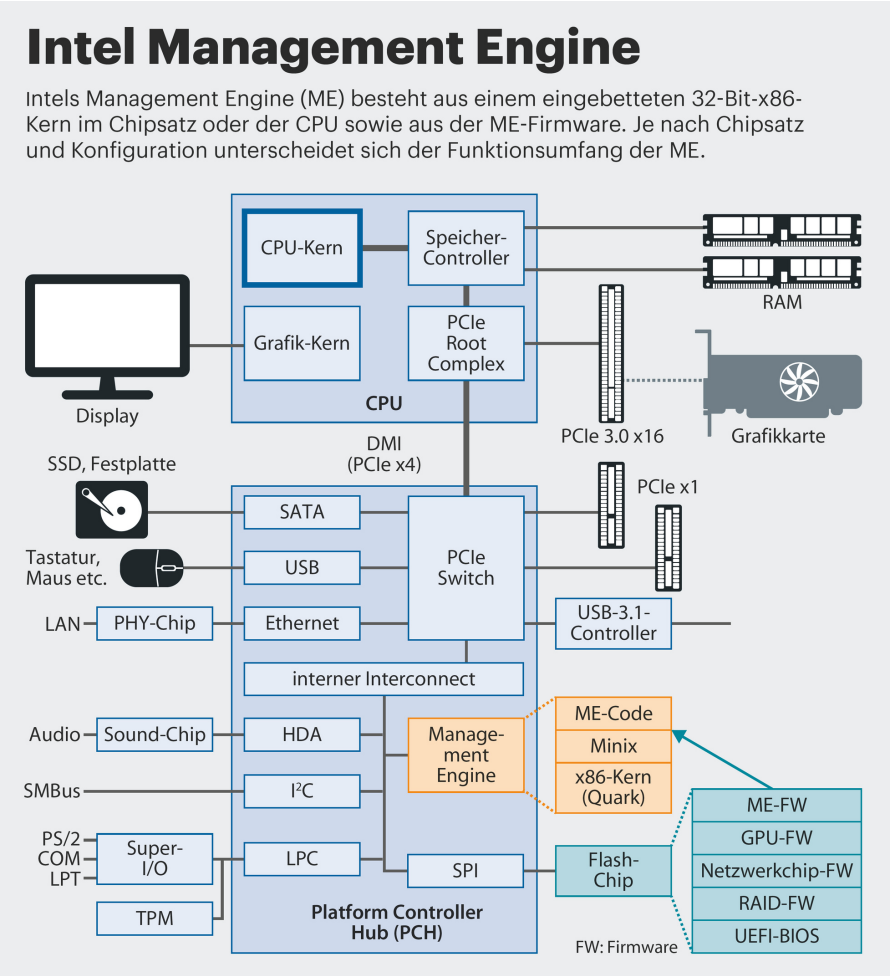


Abbildung 5: Intel Management Engine. Quelle: Heise.de

2 Definition

Wie definiert man eigentlich, was ein Betriebssystem ist, und was nicht? Dazu gibt es nicht nur eine, sondern gleich mehrere Definitionen.

2.1 Hardwareabstraktion

Man kann ein Betriebssystem als eine Hardwareabstraktionsschicht beschreiben. Das Betriebssytem ist als dafür zuständig, eine Ausführungsumgebung zu erstellen für die Programme, die darauf laufen sollen. Dazu muss das Betriebssystem den Programmen eine Abstraktionsebene bieten, damit diese nicht direkt mit der Hardware interagieren müssen. Außerdem muss das Betriebssyste die Resourcen verwalten und (idealerweise fair) zwischen den Programmen teilen.

Wenn man sich jetzt mal anschaut, wie viel Code eigentlich hinter der Ausführung eines kleinen, simplen Programms steckt, dann merkt man, wie viel eigentlich hinter den Kulissen passieren muss, damit die Programme das tun, was wir von ihnen erwarten. In Abbildung 6 ist zu sehen, welche Ebenen eigentlich unter einen Programm liegen, und wie viel Code diese Beinhalten.

Ebene	**Beispiel**	**Zeilen**
Java Bytecode		~1 000
Java Runtime	`openjdk11`	8 047 913
Support Libraries	`glib`	460 641
	`libcxx`	441 343
System Libraries	`glibc`	1 384 092
Kernel	`darwin`	1 070 917
	`openbsd`	2 235 267
	`netbsd`	5 930 334
	`linux`	17 120 205
	`windows`	~65 000 000
Hardware		

Abbildung 6: Hardwareabstraktionsebenen und Codegröße

Anmerkung
Ich habe mir mal die Freiheit genommen, die Zeilen Sourcecode für die meisten Projekte hier durchzuzählen. Dazu habe ich das Programm `cloc` benutzt (auf macOS mit `brew install cloc` installierbar). Die `libcxx` ist die C++ Standard Library, die vom Clang Compiler des LLVM Projekt genutzt wird (wird verwendet von iOS, macOS, Linux, usw.). Der macOS Kernel, `darwin`, ist Quelloffen, deswegen habe ich den auch mitgenommen. Allerdings muss man dazu sagen, dass dieser modular aufgebaut ist, und hier keine Module mitgenommen wurden, deswegen sind die Resultate nicht direkt verlgeichbar. Bei den BSD Kerneln habe ich nur den `sys/`-Ordner ausgewertet, da die anderen Ordner nicht unbedingt zum Kernel gehören (sondern Libraries, Compiler, usw. sind). Außerdem beinhalten die Angaben von Windows mehr als nur den Kernel, auch diese Angaben sind also nicht vergleichbar.

Außerdem musste man ja.

2.2 Kernel und Systemlibraries

Man könnte ein Betriebssystem so definieren, indem man sagt, dass es die Summe aus Kernel und Systemlibraries ist. Dazu müsste man aber definieren, was Systemlibraries sind. Dazu gibt es netterweise einige Standards. Die wichtigsten und interessantesten sind POSIX, was für *Portable Operating System Interface* steht, und LSB, was für *Linux Standard Base* steht.

Diese Standards existieren, weil man eine Kompatibilität zwischen verschiedenen Betriebssystemen herstellen möchte. Das bedeutet, dass man sein Programm einmal schreiben kann, und es theoretisch auf unterschiedlichen Betriebssystemen laufen lassen kann. In Tabelle 1 sieht man eine Liste von Betriebssystemstandards,

Standard	**Betriebssysteme**
POSIX	macOS (Zertifiziert seit 10.5 Leopard)
	Solaris
	Android
	Windows (nicht Out-of-the-Box, aber mit Cygwin, MinGW oder neuerdings dem *Windows Subsystem for Linux*)
LSB	Debian (teilweise)

Tabelle 1: Betriebssystemstandards

und die Betriebssysteme, die zumindest halbwegs konform sind. Wie man sehen kann, können und werden auch teilweise sehr wilde Architekturen (Windows) konform gemacht, weil dadurch viele Vorteile entstehen können, in diesem Fall Zugriff

auf Open-Source Libraries und Tools, die auf Linux oder macOS laufen.

2.3 Kernel-Mode

Wenn man schon Kernel und Systemlibraries zusammen ein *Betriebssysten* nennt, was ist denn dann bitte ein Kernel? Typischerwise würde man sagen, dass man als Kernel all das bezeichnet, was auf dem Prozessor im *Kernel-Mode* läuft.

Viele Prozessoren, unter anderem x86-CPUs, haben ein „Ring"-System, mit dem der Zugriff auf privilegierte Instruktionen limitiert wird. Der Kernel läuft bei einem solchen System im Ring 0 (der Kernel-Mode), während reguläre Programme im Ring 3 laufen (dem User-Mode).

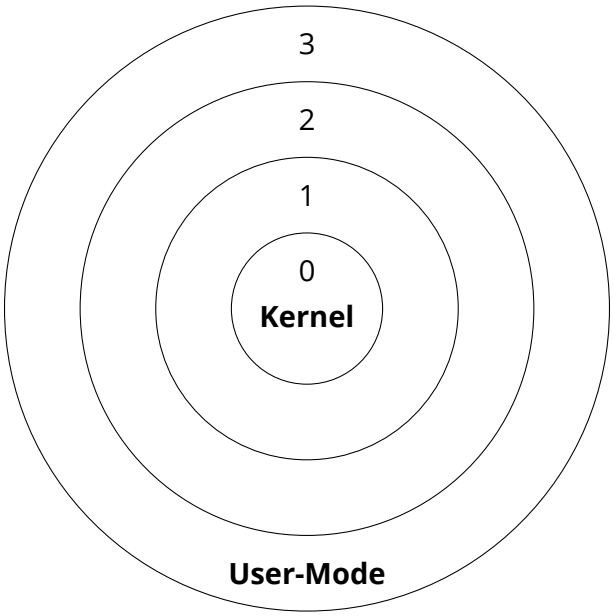


Abbildung 7: Das Ring-System bei x86 Prozessoren.

Reguläre Anwendungen haben somit also keinen direkten Zugriff auf die meisten Systemresourcen. Programme können nicht direkt auf die Festplatte zugreifen, sondern müssen den Kernel nett darum bitten, dies doch bitte für sie zu tun.

Diese Definition ist leider nicht immer nützlich, denn auf manchen Systemen gibt es diesen Unterschied zwischen privilegiertem und regulärem Code nicht. Aber das wirft auch eine andere Frage auf, was für Funktionalität packen wir eigentlich in den Kernel, und was nicht?

2.4 Monolithische- und Mikrokernel

Es gibt zwei Arten von Kerneln, die man Unterscheiden kann; diese sind *Monolithische-* und *Mikrokernel*. Der hauptsächliche Unterschied unter den beiden ist wie viel Code

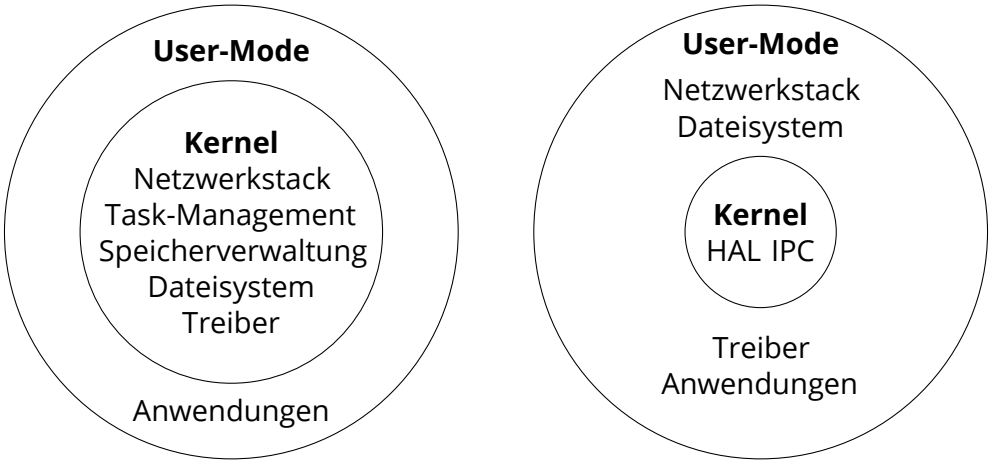


Abbildung 8: Übersicht über die Architektur Monolithischer- und Mikrokernel.

im Kernel-Mode läuft. Abbildung 8 zeigt, wie die Architektur in etwa aufgebaut ist bei den verschiedenen Typen von Kerneln.

Es gibt unterschiedliche Gründe, Monolithische oder Mikrokernel zu verwenden. Die meisten Kernel, die in Endnutzersystemen benutzt werden (macOS, Linux, Windows), sind Mikrokernel. Das liegt einfach daran, dass Mikrokernel generell schneller sind. Abbildung 9 zeigt, wie viele CPU Zyklen gewisse Operationen brauchen. Dort sieht man, dass ein Kernelaufruf (also ein Switch vom User-Mode in den Kernel-Mode und wieder zurück) stolze 1000 bis 1500 CPU-Zyklen braucht. Warum ist das so? Wenn der CPU einen solchen Kontextswitch macht, müssen viele CPU-Internen Datenstrukturen zurückgesetzt werden, zum Beispiel den *Translation Lookaside Buffer*, weil der Kernel eine andere Sicht auf den Speicher hat. Der Status des CPUs muss gespeichert werden (also alle Register, wenn der Kernel Gleitkommazahlberechnungen machen muss, dann müssen auch alle Floating-Point-Register gespeichert werden) damit der CPU nach dem Systemcall in den selben Zustand versetzt werden kann, wie er davor war. Außerdem müssen viele Caches weggeworfen werden. Da Switches zwischen Kernel- und User-Mode so teuer sind, versucht man, so viel wie möglich direkt im Kernel zu machen.

Es gibt aber auch Nachteile bei einer solchen Architektur. Alles, was im Kernel läuft, ist ein potenzielles Sicherheitsproblem, denn es hat uneingeschränkt Zugriff auf alles. Wenn ein Programm abstürzt, kann es dem System nicht schaden, weil alle Zugriffe auf Resources vom Kernel abgefangen werden. Wenn ein Treiber abstürzt, gibt es aber kein Sicherheitsnetz. Daher kommen Bluescreens (bei Windows) oder Kernel Panics (bei Linux).

Die Mikrokernelarchitektur hat einen anderen Ansatz. Hier versucht man, so *wenig* wie möglich im Kernel-Mode laufen zu lassen. Man macht hier nur das absolut notwendige: man bietet eine Hardwareabstraktionsschicht und einen Mechanismus, mit dem Prozesse kommunizieren können.

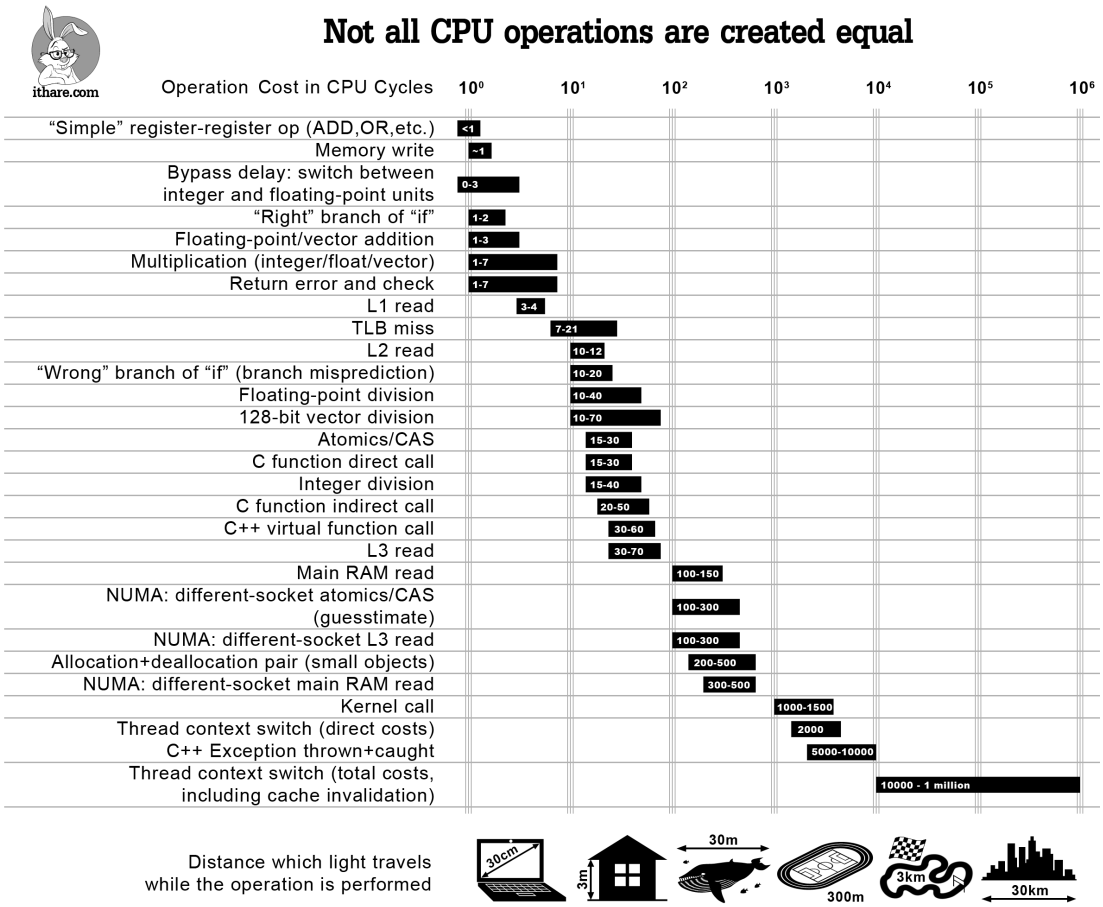


Abbildung 9: Übersicht über die Kosten verschiedener Operationen.
Quelle: http://ithare.com.

Die Idee dahinter ist, dass man Funktionalität, die sonst im Kernel wäre, einfach im Userspace laufen lässt. Treiber laufen dann als reguläre Prozesse, und Anwendungen können mit IPC Anfragen stellen. Die Vorteile hier sind Sicherheit und Stabilität, denn nur das Allernötigste läuft mit maximalen Rechten. Wie man vorhin gesehen hat, hat Apple in iPhones einen Subprozessor, die *Secure Enclave*, die unter anderen Verschlüsselungskeys verwaltet. Diese nutzt einen Mikrokernel, genauso wie das Merkel-Phone und gewisse andere Sicherheitsrelevante Produkte.

2.5 Koordination

Ein Betriebssytem kann man auch Ressourcenverwaltung ansehen. Es verwaltet die Ressourcen, die einem System zur Verfügung stehen. Dazu gehören zum Beispiel Speicher, Rechenleistung und Peripheriegeräte. Das Betriebssystem sorgt dafür, dass jedes Programm die Ressourcen effizient und fair nutzen kann.

Das Betriebssystem kontrolliert die Ausführung von Programmen, um Fehler und falsche Verwendung von Ressourcen zu verhindern.