Wie schreibt man ein Betriebssystem?

Vom BIOS in den Userspace

Andreas Galauner
Easterhegg 2011

Democode

- Es gibt Democode:
  - http://github.com/G33KatWork/SigInt10OSWorkshop
  - git clone git://github.com/G33KatWork/SigInt10OSWorkshop.git
- Am besten direkt clonen und make eingeben, Toolchain (binutils, gcc, nasm, qemu) wird gebaut
- „make V=1“ macht verbose output, falls es Probleme gibt
- Happy Hacking!

- Was ist ein Betriebssystem
- Architekturen
- Allgemeine x86 Einführung
- 32 Bit Modus
- Virtueller Speicher
- Interrupts und Exceptions
- 64 Bit Modus
- Multitasking

Betriebssysteme

Was ist ein Betriebssystem?

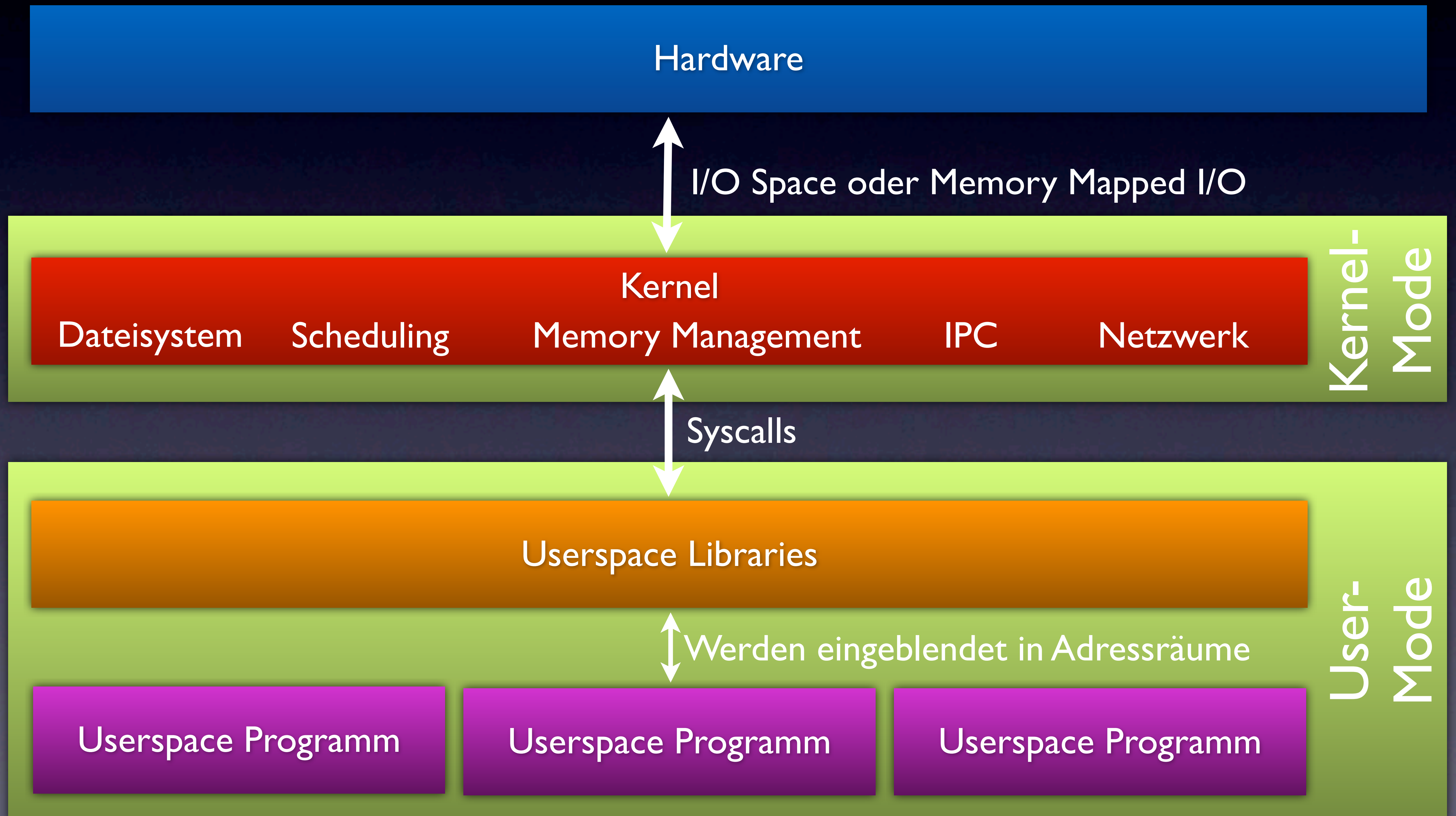
- Etwas geekiger:
  - Abstraktion der Hardware
  - Schaffung einheitlicher Schnittstellen
  - Gerechte Verteilung zur Verfügung stehender Resourcen unter den „Anwärtern“

Woraus besteht ein Betriebssystem?

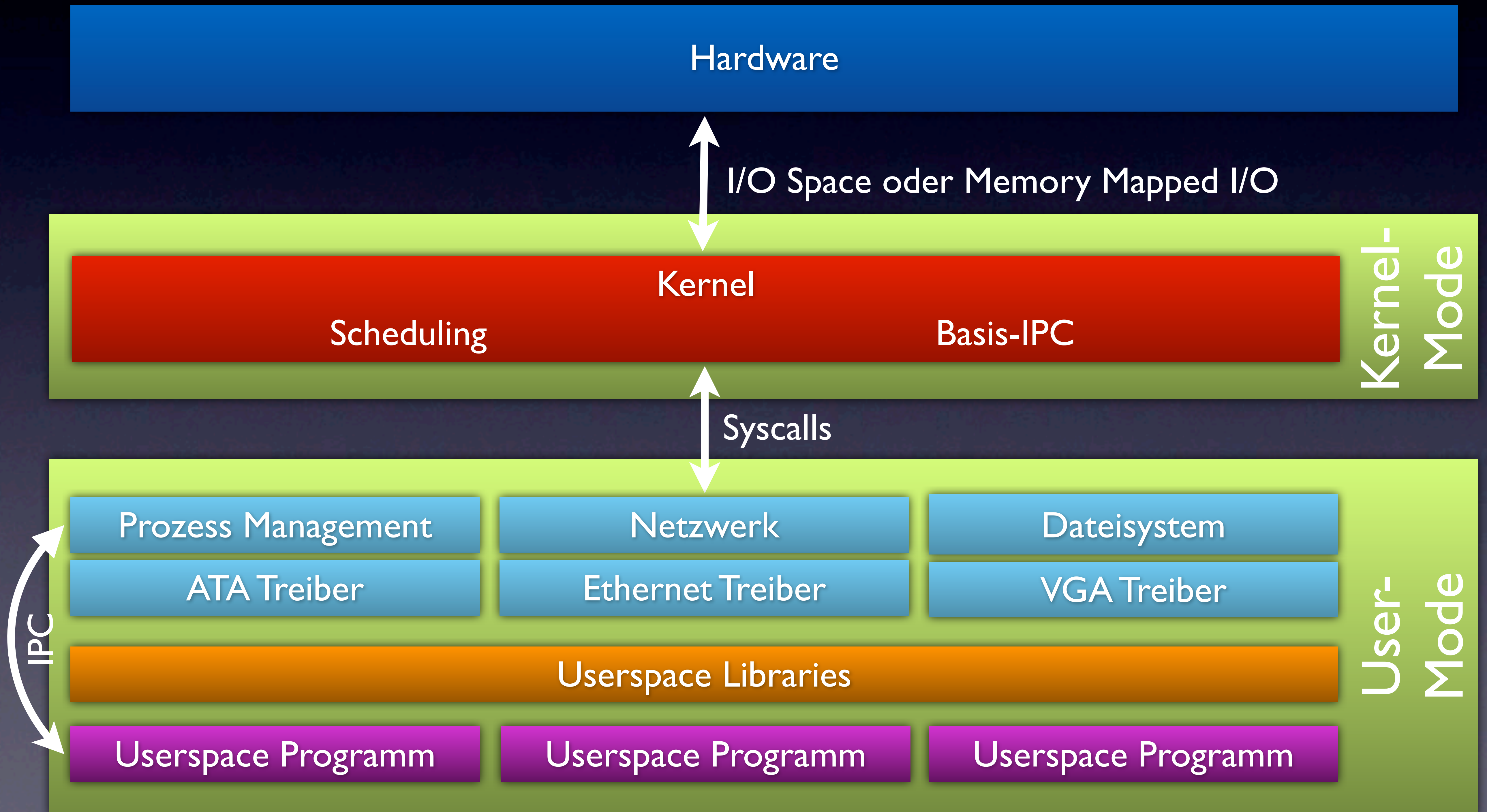
- Kernel, evtl. mit abgespaltenem HAL
- Userspace Libraries
- Userspace Services
- Userspace Applikationen

Betriebssystemarchitekturen

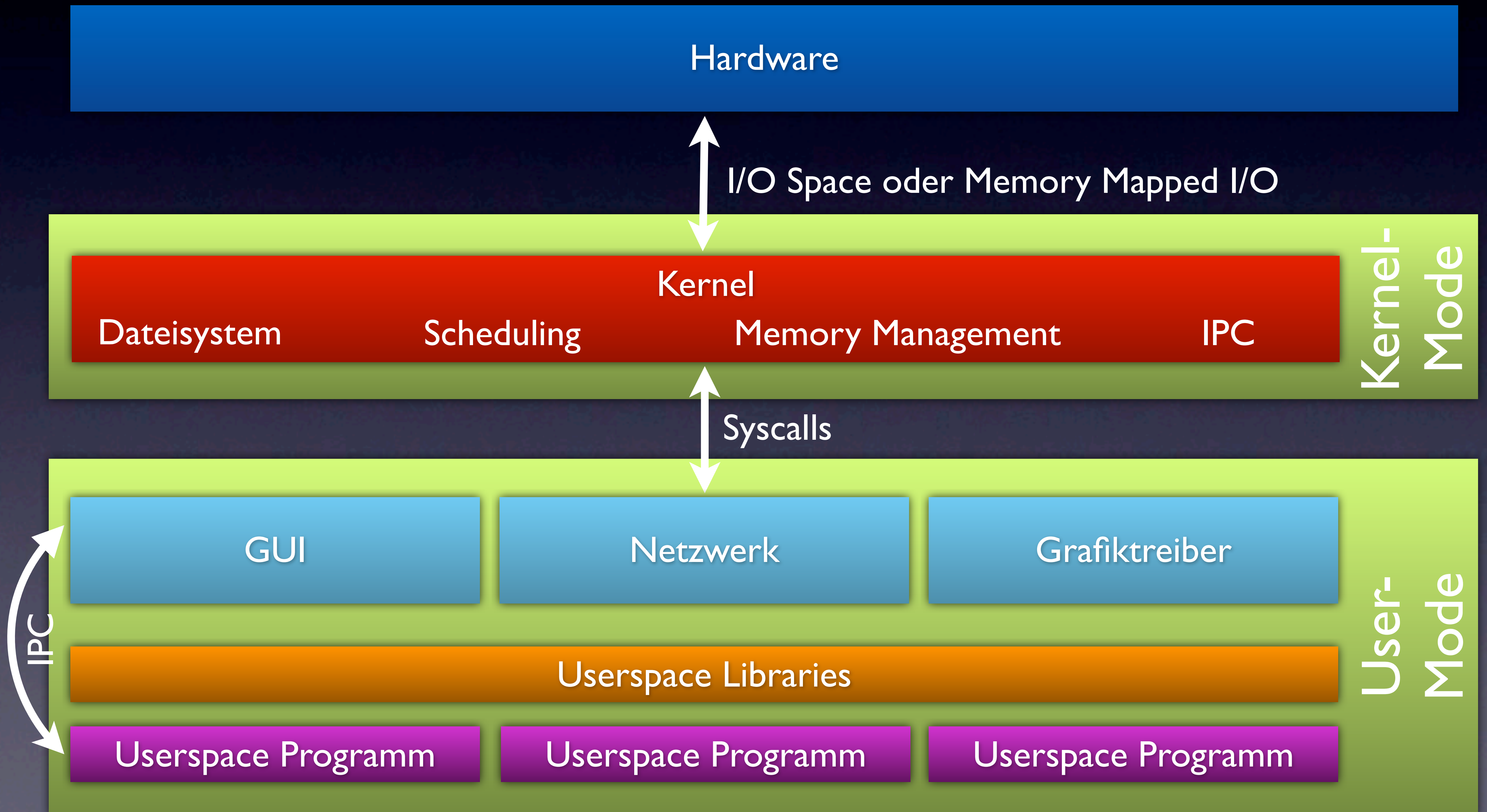
Monolithischer Kernel



Microkernel

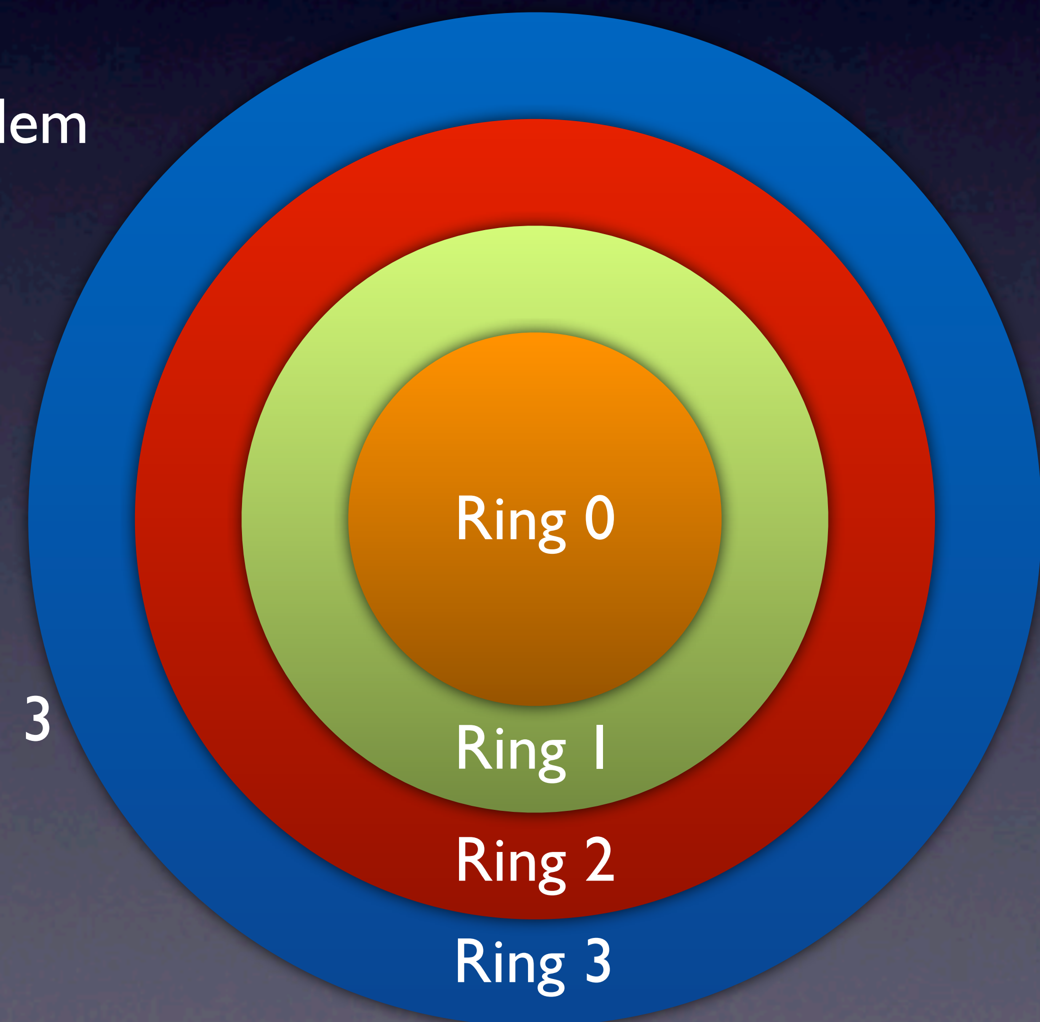


Hybridkernel



Privilege Levels

- Einführung mehrerer Ebenen in Hardware
- Je niedriger die Ebene, umso mehr Rechte hat sie
- IA-32 hat 4 Ringe
- Instruktionen werden abhängig vom Ring, in dem ein Prozess läuft, ausgeführt oder blockiert
- Kernel darf alles
- Ring 0 ist Kernel, Ring 3 Userspace
- Userspace ist stark limitiert: Kein direkter Hardwarezugriff, etc.
- Betriebssysteme nutzen meist nur Ring 0 und 3 aus Gründen der Kompatiblität zu anderen Architekturen



Syscalls

- Wechsel von höheren Ringen in tiefere nicht ohne weiteres möglich, ist Sinn der Sache
- Ist aber nötig, um Kernelcode vom Usermode aus aufzurufen
- Wird über Syscalls geregelt
- Verschiedene Möglichkeiten: Trap, Software Interrupt, Sysenter/Sysexit
- Beispiel: open, mmap, fork, Berkley-Sockets
- Rootkits würden sich in die Syscalls klemmen und Ergebnisse „passend“ modifizieren

Allgemeine x86 Einführung

Intel x86 Geschichte

- Mikroprozessor-Architektur
- Entworfen von Intel
- Erstmals im 8088/8086 im Jahre 1978 genutzt
- 8086 war 16 Bit
- Nachfolgemodelle: 80186, 80286, 80386 etc.
- 32 Bit ab 80386
- Virtueller Speicher mit MMU ab 80486
- AMD entwickelte x86_64 (EM64T bei Intel)

Bootvorgang

- Ab 80386 startet Prozessor bei 0xFFFFFFF0 (reset vector)
- BIOS liegt aber in unteren 64KB (ca.)
- Lösung: Nach Reset wird BIOS vom Chipsatz meistens in die obersten paar MB gemappt
- Wenn BIOS läuft, wird Black-Initialisierungs-Voodoo ausgeführt (näheres dazu unter coreboot.org)
- Danach werden, je nach Bootdevice, 512 Byte nach 0x0:07C0 kopiert und hingesprungen, falls Bootsignatur in letzten zwei Byte stimmt (0x55AA oder 0xAA55 - Big Endinan oder Little Endian)
- CPU befindet sich hier im Real Mode (8086, 16 Bit)

Segmentierung

- 8086 hatte 20 Adressleitungen (A0 - A19)
- Physikalisch adressierbar waren somit 1MB RAM
- Register sind aber alle nur 16 Bit breit, womit 4 Bit im Adressbus brach lägen
- Segmentierung löst das Problem

Segmentierung

- Speicher wird in 16-Byte große Segmente (16 Byte = 64KB adressierbar) aufgeteilt
- Segmente können sich überlappen
- In 16 Bit Registern (CS, DS, ES, SS) kann für verschiedene Zwecke ein Segmentanfang ausgewählt werden
- Physikalische Adresse resultiert aus:
  - Segmentregister * 16 + Offset

Real Mode

- Ist der „alte“ 16 Bit Modus eines 8086
- Intel ist voll binär Abwärtskompatibel
- Bootloader starten hier
- BIOS bietet Systemfuntkionen über Interrupts an (Floppy steuern, Text ausgeben...)
- DOS erweiterte diese Routinen früher und hookte einige der BIOS-Funktionen

32 Bit Modus

Protected Mode

- Mit 80286 kam ein 24 Bit Adressbus (16MB) und der Protected Mode
- Alle Register blieben 16 Bit breit
- Der Protected Mode bietet Speicherschutz durch manuelles erstellen der Speichersegmente (GDT - Global Descriptor Table und LDT - Local Descriptor Table)
- Insgesamt gab es 1GB virtuellen Speicher

A20 Gate

- Es gab DOS-Programme, die sich darauf verließen, dass es nur 20 Bit breite physikalische Adressen gab
- Sie nutzten also explizit den Überlauf der Register, um $stuff im Speicher zu adressieren
- Der 286er konnte aber mehr als 1MB Speicher adressieren und sollte abwörtskompatibel sein
- IBM schuf die „Möglichkeit“ die 20. Adressleitung fest auf GND zu legen
- Somit war der Überlauf wieder da, es gab allerdings auch nur 1MB adressierbaren Speicher

A20 Gate

- Existiert heute immernoch
- Bereitet einem viel Freude
- Wurde früher vom Keyboard Controller (SRSLY!) gesteuert
- Heute über Chipsatz
- Es gibt gefühlte 3 mio. Arten das Ding zu aktivieren

Umschalten in 32 Bit Protected Mode

- Nachdem wir jetzt mehr als 1MB Ram nutzen können, wollen wir 32 Bit
- Segmentierung ist fies, 16 Bit sowieso
- Blos weg da
- Mit 80386 kam der 32 Bit Protected Mode
- Wie Protected Mode aus 80286, nur volle 32 Bit - und zwar überall. IA-32 Architektur

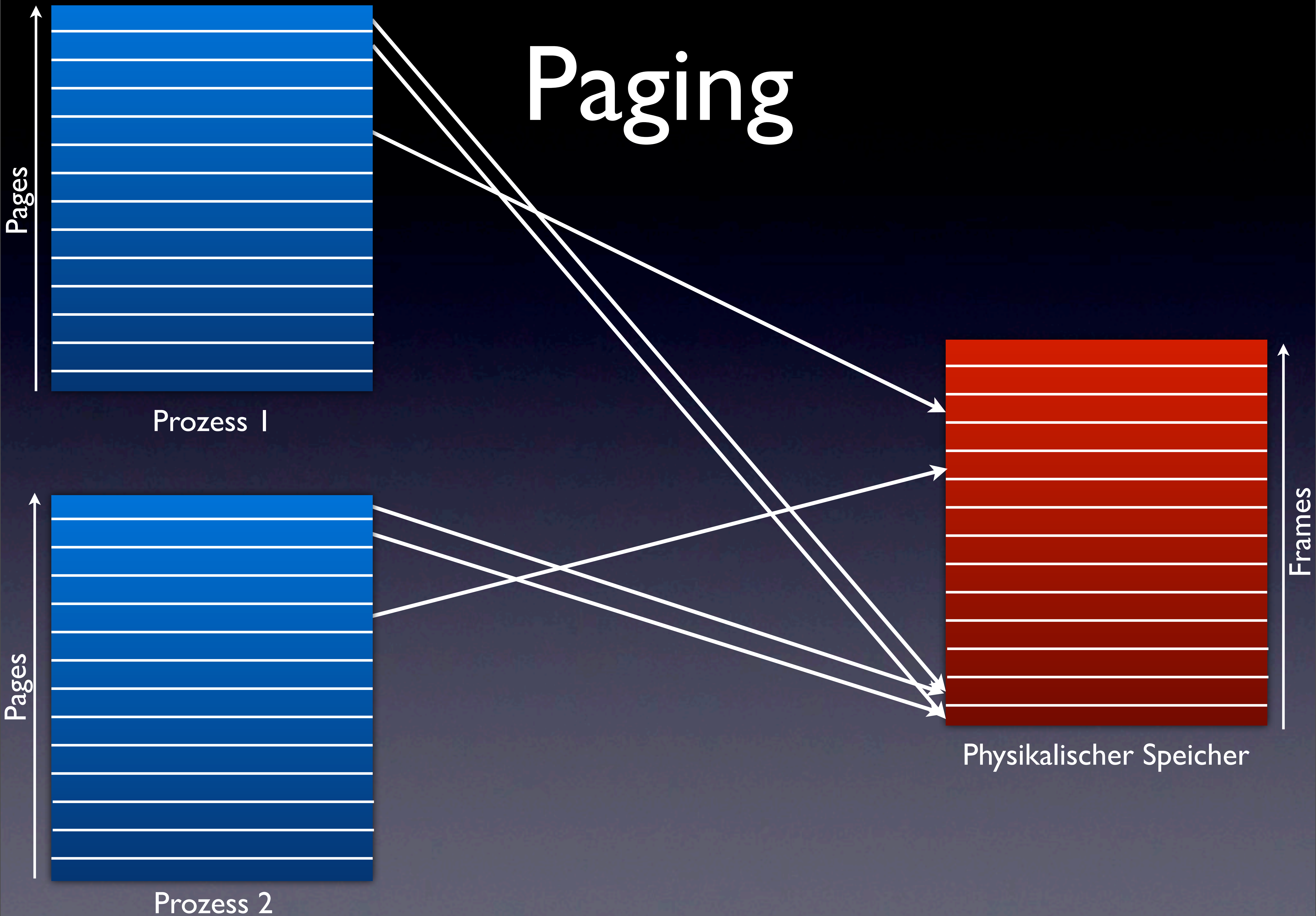
Umschalten in 32 Bit Protected Mode

- Was müssen wir dafür tun?
- Segmentierung konfigurieren
  - Möglichst so, das wir ein flaches, 4GB großes Segment haben, da wir dieses später eh wieder über Paging unterteilen
  - Mittels GDT
- Danach im CR0 Register das 0. Bit setzen
- Mit einem Far-Jump in neues, über GDT beschriebenes, Speichersegment springen
- Nun stehen uns keinerlei Hilfen vom BIOS mehr zur Verfügung

Virtueller Speicher

Virtueller Speicher

- Einer der Grundpfeiler zur Abschottung einzelner Prozesse
- Jeder Prozess erhält einen eigenen Adressraum
- Der Adressraum kann dann via Paging realer Speicher zugewiesen werden
- Virtueller Speicher ermöglicht setzen einiger Kontrollbits (Privilege Level, NX-Bit, Read/Write etc.)
- Bei Missachtung dieser Kontrollbits, wird eine Exception geworfen: Page Fault
- Adressumsetzung passiert transparent in der CPU durch die Memory Management Unit (MMU)
- Auch andere Inhalte im Speicher möglich, wie z.B. Dateien
- Optimierung des Speicherverbrauchs durch gleichzeitiges Mappen einzelner Frames in mehrere Adressräumen und evtl. Copy-On-Write
- Swapping


Paging
Pages
Prozess 1
Pages
Prozess 2
Frames
Physikalischer Speicher

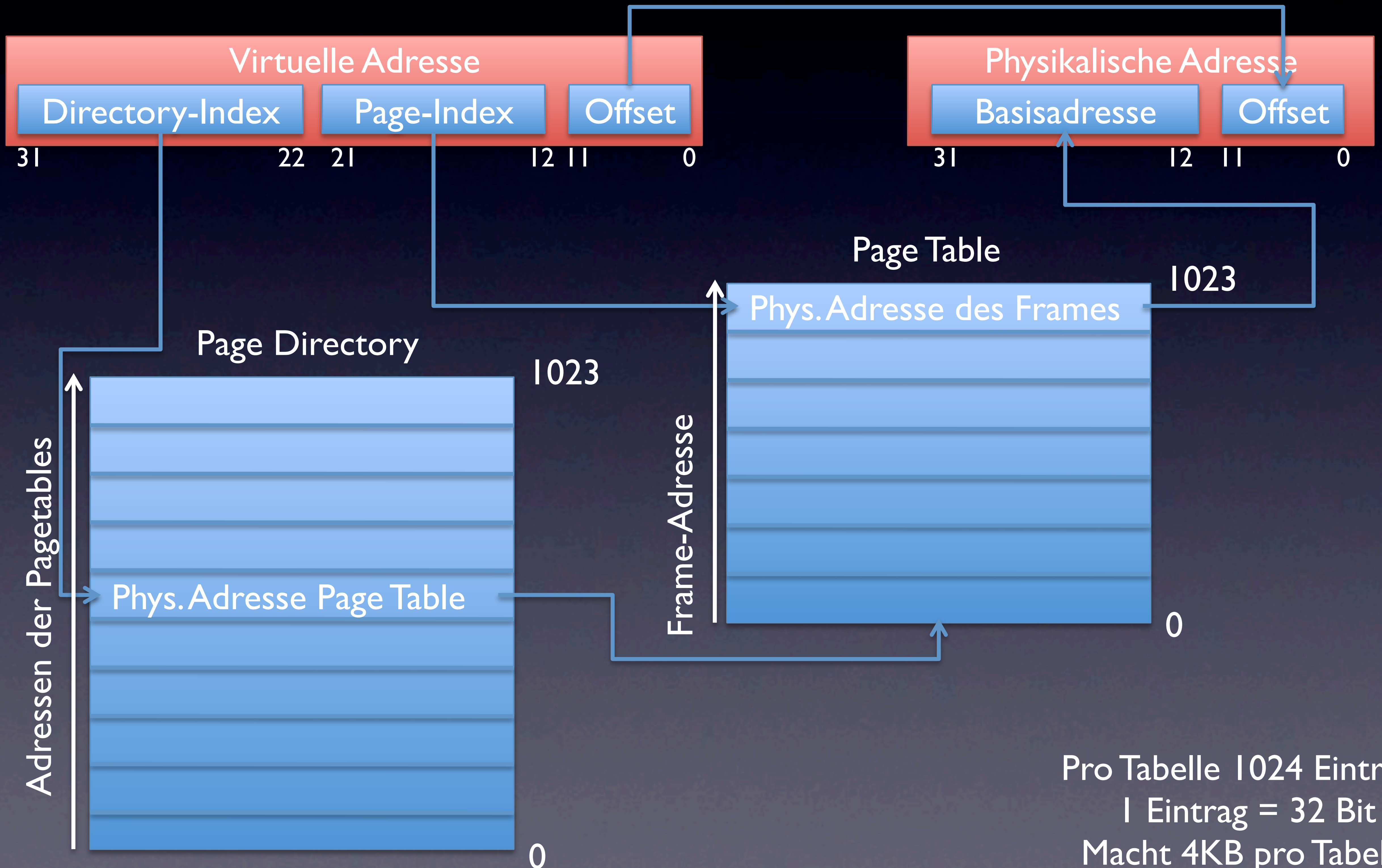
Paging

- Wir unterteilen den physikalischen und virtuellen Speicher in gleich große Teile
- Physikalische Teile nennen wir „Frames“
- Virtuelle Teile werden die „Pages“
- Jeder Teil ist bei x86 (IA-32) 4KB oder 4MB groß
- Bei 4KB haben wir also 4GB / 4KB = 1048576 Pages, die wir nun mit Frames befüllen können
- Ein Page Directory (Mapping von Pages zu Frames) ist also (bei 4B langen Pointern) 1048576 * 4B = 4MB groß → zu groß!

Paging

- Lösung für unser Größenproblem: 2-stufiges Paging
- Obere 10 Bit der virtuellen Adresse ergeben Index im Page Directory, der auf eine Page Table zeigt
- Folgende 10 Bit ergeben den Index der Page in der Page Table
- Letzte 12 Bit sind dann Offset im Frame selbst

Paging



Pro Tabelle 1024 Einträge
1 Eintrag = 32 Bit
Macht 4KB pro Tabelle

Paging

- Aktivierung?
- Page Directory und Page Tables von Hand zusammenbasteln
- Page Directory Adresse in CR3 schreiben
- Bit 31 in CR0 setzen
- Vorsicht: Vom Linker erstelle Adressen stimmen unter Umständen nicht mehr
- Code muss entweder relokiert werden
- Oder kleineres Programm vorher initialisiert Paging und springt dann in den eigentlich und korrekt auf die neue Basisadresse gelinkten Kernelcode

Interrupts und Exceptions

Interrupts

- x86 kann auch Interrupts
- Prozessor hat 256 verschiedene Interrupts
- Unterste 32 sind als Exceptions reserviert
- Prozessor hat (oder hatte... virtualisierter Bus usw.) eine physikalische Interruptleitung
- An Prozessor hingen 2 PICs (Programmable Interrupt Controller), die Interrupts angenommen haben und an der Prozessorleitung gezogen haben
- Im Anschluss konnte man die PICs fragen, was los war

PIC

```
PIC 1 (Master)

+--------------+                           PIC 2 (Slave)
| IRQ  0       + <--- Timer                +-------------+
| IRQ  1       + <--- Tastatur             | IRQ  8      + <--- Echtzeituhr
| IRQ  2       +------------------+        | IRQ  9      + <--- ...
| IRQ  3       + <--- Seriell     |        | IRQ 10      + <--- ...
| IRQ  4       + <--- Seriell     |        | IRQ 11      + <--- ...
| IRQ  5       + <--- Soundkarte  |        | IRQ 12      + <--- PS/2 Maus
| IRQ  6       + <--- Floppy      |        | IRQ 13      + <--- Koprozessor
| IRQ  7       + <--- Parallel    |        | IRQ 14      + <--- Festplatte
+------+-------+         Port     |        | IRQ 15      + <--- Festplatte
       |                          |        +-----+-------+
       |                          |              |
       |                          +--------------+
       |
       +--->>> Zur CPU
```

Interrupts handlen

- Um Interrupts zu behandeln müsen wir folgendes tun:
- IDT (Interrupt Descriptor Table) erstellen
- IDT laden
- Interrupts aktivieren
- Für Hardware-Interrupts MUSS der (A)PIC konfiguriert werden

64 Bit Modus

Long Mode

- Long Mode ist der 64 Bit Modus
- Aktivierung:
  - In Protected Mode wechseln, Paging aus
  - Paging-Tabellen basteln
  - PAE in CR4 aktivieren (max. 64GB physikalischen RAM)
  - Longmode in MSR (Model Specific Register) aktivieren
  - Paging und Longmode in CR0 aktivieren
  - 64 Bit GDT laden

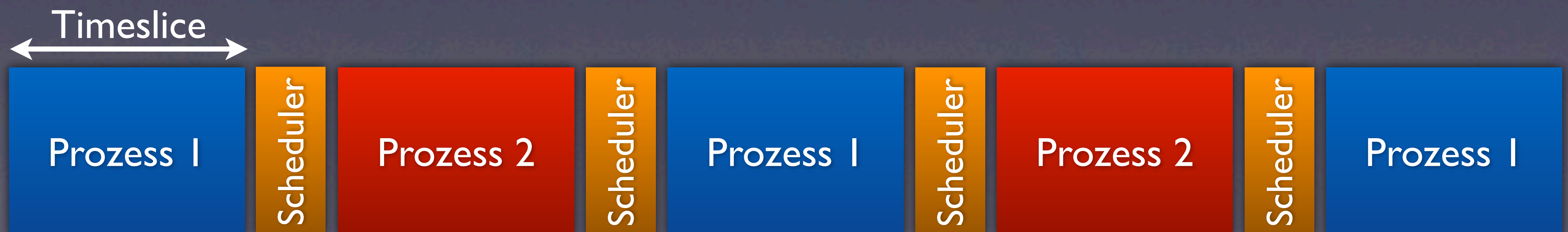
Multitasking

Multitasking

- Jeder Task bekommt einige ms CPU-Zeit, danach wird gewechselt
- Umschalten läuft völlig transparent ab
- Kompletter State eines Tasks muss gesichert und neuer geladen werden
- Unterbrechungen finden durch Timer Interrupt statt

Scheduling

- Unterteilung der Zeit in Zeitschlitze
- Zuweisung einzelner Zeitschlitze an Prozesse = CPU-Zeit
- Zeitschlitz wird periodisch durch Timer Interrupt unterbrochen
- Auswahl des nächsten Prozesses kann anhand von Prioritäten, Wartezuständen etc. geschehen
- Scheduler übernimmt, je nach Verfahren, auch die unterste Schicht des IPC (z.B. Signal Handling unter UNIX)



Task State

- x86 kann Hardware Task Switching
- Wir machen es in Software
  - Gründe:
  - Kaum Geschwindigkeitsnachteile
  - Portabler
- Zum State gehört:
  - Register
  - Stack
  - evtl. Page Directory

State sichern

- Die CPU führt fröhlich Instruktionen aus und es passiert ein Interrupt
- Beim Interrupt werden verschiedene Sachen auf den Stack geschoben, um nachher wieder zurückkehren zu können
- Die ist ein Teil unserer zu sichernden States
- Zur Sicherheit schieben wir am Anfang des Interrupts den Rest auch noch direkt auf den Stack
- Nun brauchen wir uns den Kram nur noch vom Stack holen und in eine struct schreiben und der State ist gesichert

Neuen Task wählen

- Die Wahl des neuen Tasks kann beliebig kompliziert werden
- Auswahlkriterien können Zustand sein (wartend/schlafend, ausführend), Priorität, etc.
- Wir machen hier Round-Robin, also einfach immer den nächsten wählen ohne viele Extras

State wiederherstellen

- Um den State eines Tasks wieder herzustellen, müssen wir die Register einzeln setzen, alle erwarteten Interrupt-Rücksprungargumente auf den Stack pushen
- Und ireten
- Neuer Task sollte laufen

Kurze GeexOS Demo

Fragen?
Antworten!