Gerion Entrup
entrup@sra.uni-hannover.de
Winter 2021/2022

Betriebssystemtechnik

Aufgabe 2

Adressräume, Paging und Speicherschutz

Ziele dieser Übung

Diesmal sollt ihr Speicherschutz zwischen den Anwendungen sowie Kernelspeicher und den Anwendungen einführen. Dafür werdet ihr Freispeicherverwaltungsalgorithmen implementieren und den Anwendungen und dem Kern einen Heap zur Verfügung stellen.

Kurzfragen

- Was ist Paging? Wie unterscheidet sich Paging von Segmentierung?
- Wer setzt den Paging-Mechanismus um?
- Wer implementiert die Policy?

1 Motivation

In der Aufgabe 1 wurde der Privilegienwechsel implementiert. Wir haben dadurch eine Trennung zwischen Kern- und Usercode. Allerdings ist es für die Anwendungen immer noch möglich, gegenseitig Daten und Code auszulesen, zu modifizieren oder auf Kernelspeicher zuzugreifen. Ziel dieser Aufgabe ist es, jeden Prozess in seinen eigenen Adressraum zu verbannen und zu verhindern, dass willkürlich Daten oder Code gelesen bzw. verändert werden können. Ein Wechsel ist den Kern ist dann nur noch über einen Systemaufruf möglich. Ein weiterer Nebeneffekt ist die deutlich feingranularere Rechteverwaltung, die wir in Aufgabe 3 ausnutzen werden.

1.1 Überblick

Auf x86 werden die logischen Adressen (Zeiger + impliziter Segment Selector) zunächst durch Segmentierung in lineare Adressen übersetzt, dann durch Paging in physische Adressen. Bei der Segmentierung kann der Interrupt 13 (Segmentation Fault) auftreten, beim Paging tritt ggf. der Interrupt 14 (Page Fault) auf. Abbildung 1 zeigt den Ablauf der Adressumsetzung auf einem x86-Prozessor schematisch.

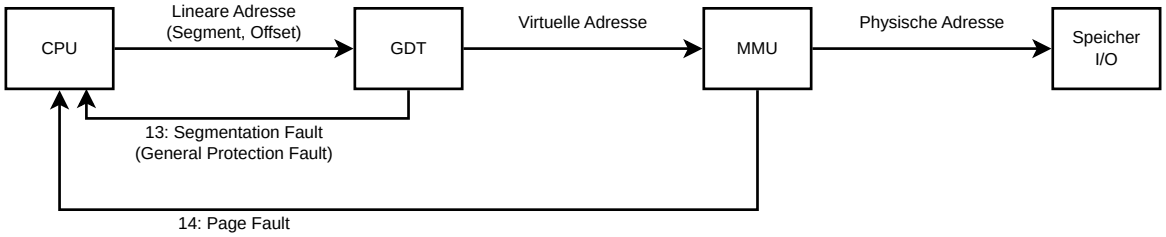


Abbildung 1: Adressumsetzung vom x86-Prozessor

2 Paging

Beim Paging wird der Speicher in sogenannte Pages (im linearen Adressraum), bzw. Frames (im physischen Adressraum) zerlegt. Eine Page bezeichnet ein in linearen Adressen zusammenhängendes Stück Speicher gleicher Größe (eine Zweierpotenz), ein Frame ist das Pendant im physischen Adressraum. Lineare Adressen werden anhand von Paging zu physischen Adressen übersetzt, die zum Abrufen von Werten aus dem Speicher benutzt wird. Paging übersetzt somit eine 32 bit Zahl in eine andere 32 bit Zahl und bildet damit beim Zugriff von einer Page auf einen Frame ab.

Auf x86 gibt es mehrere Modi für die Adressumsetzung mit Pages. Wir verwenden den 32-Bit-Modus (IA32). Die Paging-Datenstruktur besteht aus

- dem Page Directory und
- den bis zu 1024 Page Tables

Ein Page Directory hat 1024 Einträge und verwaltet einen 4 MiB Speicherbereich. Wird der Speicherbereich in 4 KiB Pages eingeteilt, wird aus dem Page Directory-Eintrag auf eine Page Table verwiesen, die jeweils die Übersetzung auf 4 KiB Granularität vornimmt. Alternativ ist es möglich, die Page Table wegzulassen und direkt auf einen 4 MiB Block als zusammenhängenden Speicherblock zu übersetzen. Da wir aber in fast allen Fällen eine feingranularere Teilung brauchen, benutzen wir normal ein zweistufiges Paging-System. Die Einträge sind in Abbildung 3 dargestellt.

Zwischenfragen

- Gibt es Nachteile von Paging? Warum könnten manche Betriebssysteme es nicht verwenden?[1]
- Wir brauchen eine Übersetzung einer 32 bit-Zahl in eine andere 32 bit-Zahl? Welcher Mechanismus ist dafür sinnvoll?

2.1 Adressübersetzung

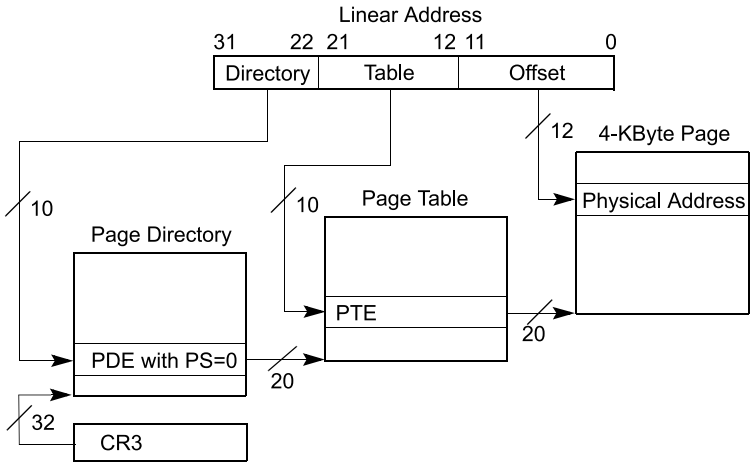


Abbildung 2: Adressübersetzung mit 4 KiB Pages [3, Fig 4-2]

Bei der Übersetzung der linearen Adresse (nach der Segmentierung) in die physische Adresse wird diese zunächst in drei Teile zerlegt. Mit den 10-most-significant Bits wird der Eintrag im Page Directory gefunden. Das Page Directory muss somit auf 4 KiB ausgerichtet sein (warum?). Die Basis-Adresse des

[1] Es ist unglaublich langsam!

Page Directories steht im `CR3`-Register. Ist das Granularitäts-bit im Page Directory Eintrag gesetzt, verweist die Basis-Adresse auf eine Page Table (ebenfalls 4 KiB aligned), andernfalls auf einen Block von 4 MiB Speicher. Der Eintrag in die Page Table wird mit den mittleren 10 bit identifiziert. Dieser enthält die Basisadresse, die den Start des physischen Speicherblocks anzeigt. Die übrigen 12 bit der linearen Adresse geben einen Offset innerhalb der Page an. Abbildung 2 zeigt die Übersetzung schematisch.

2.2 Paging Datenstruktur

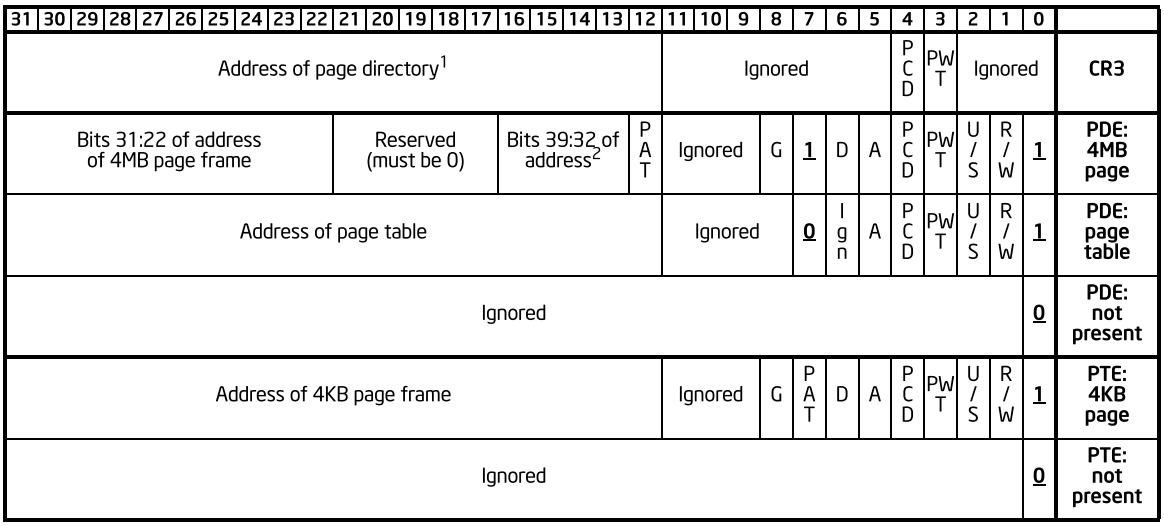


Abbildung 3: Aufbau der Page Directory Entries und Page Table Entries [3, Fig 4-4], vgl. [2]

`P`	(Bit 0, LS)	liegt ein gültiger Eintrag vor? Falls nicht, führt ein Zugriff auf eine lineare Adresse im Bereich der Page zum Page Fault.
`R/W`	(Bit 1)	Read/Write, setzt die Page auf schreibbar
`U/S`	(Bit 2)	User/Supervisor, setzt die Page auf zugreiffbar durch Nutzercode
`PWT`	(Bit 3)	Page-level Write-Through, aktiviert Write Through
`PCD`	(Bit 4)	Page-level Cache Disable (deaktiviert Caching für die Page)
`A`	(Bit 5)	Accessed, wurde auf die Page zugegriffen? (Wird nicht von der CPU aktualisiert)
`D`	(Bit 6)	Dirty, wurde die Page verändert? (wird nicht zurückgesetzt von CPU)
`S`	(Bit 7)	Size, falls gesetzt beschreibt der Page Directory Eintrag einen 4 MiB Speicherbereich[2]
`PAT`	(Bit 7)[3]	
`G`	(Bit 8)	Global, falls gesetzt, führt dazu, dass der TLB-Eintrag nicht gelöscht wird, wenn `CR3` aktualisiert wird.
`Address`	(22:31)	Zeiger auf 4 KiB aligned Page Tables bzw. Pages

Zwischenfragen

- Wie groß ist ein Page Directory und eine Page Table?
- Sind 4 KiB Pages sinnvoll? Warum?
- Wie hoch ist der maximale und minimale Speicherverbrauch der Datenstrukturen (also um den kompletten 32 bit-Adressraum zu mappen)?[4]

[4] Lösung: $(1 \cdot \text{PD} + 1024 \cdot \text{PT}) \cdot 4\,\text{KiB} = 4100\,\text{KiB}$, minimal: $(1 \cdot \text{PD} + 1 \cdot \text{PT}) \cdot 4\,\text{KiB} = 8\,\text{KiB}$

- Wie hoch ist der Speicherverbrauch bei einem flachen Mapping, also der direkten Übersetzung durch eine flache Tabelle?

2.3 TLB

Paging ist im Vergleich zu einem direkten Speicherzugriff unglaublich langsam. Darum gibt es mit dem Translation Lookaside Buffer (TLB) einen Cache, der Schlüssel-Wert-Paare zwischen virtueller und physischer Adresse zwischenspeichert. Diesen Cache kann (und muss) man invalidieren:

`mov %eax, %cr3`:	Alle Einträge löschen (TEUER!).
`invplg (%eax)`:	Invalidiert genau den *einen* Eintrag, der die Adresse, die in `%eax` steht, enthält.

Zwischenfragen

- Wie viele Speicherzugriffe sind notwendig, um ein Byte zu lesen?
- Wann muss der Cache invalidiert werden?

3 Speicherlayout

Wir kommen jetzt zu einem Punkt, bei dem wir Objekte dynamisch allozieren möchten. D. h., dass wir uns erstmals darüber Gedanken machen müssen, wo überhaupt freier physischer Speicher ist und wie wir Anwendungsspeicher dynamisch allozieren können.

3.1 Anwendungen laden

Bislang haben wir Anwendungen statisch zusammen mit dem Kern in ein Image einkompiliert. Wir wollen aber dahin kommen, diese dynamisch nachzuladen. Normalerweise geschieht dies über einen Festplattentreiber, auf den wir hier aber verzichten. Wir benutzen stattdessen eine `initrd`, mit der dem Kern relativ einfach beim Booten Daten mitgeben können. Die `initrd` ist dabei ein zusätzliches Stück Speicher, das beim Booten vom Bootloader zusätzlich zum Kernel-Image in den physischen Speicher geladen wird. Der Bootloader übergibt anschließend dem Betriebssystem die Adresse der `initrd` als Teil des Multiboot-Headers, den wir im nächsten Kapitel genauer beschreiben.

3.2 Freien physischen Speicher finden

Der freie verfügbare Speicher kann über das BIOS etc. ausgelesen werden, was relativ umständlich ist.

Glücklicherweise nimmt uns aber der Bootloader, der auch unser StuBSmI lädt, die Arbeit ab, indem er uns über eine Liste sagt, wo freier physischer Speicher liegt. Diese Informationen werde als Teil des Multiboot-Headers [1] übergeben, der allgemeine Informationen des (Multiboot kompatiblen) Bootloaders an das Betriebssystem enthält. Das Format des Headers ist dabei standardisiert. Wir müssen ihn aber trotzdem noch auslesen und parsen.

Die Flags zu Beginn des Headers geben an, welche Attribute valide sind. Von Bedeutung sind für uns die `mods` und die `mmap`-Attribute. Die Module geben an, welche Speicherabbilder vom Bootloader wohin geladen worden sind. Vor allem ist dort auch unsere `initrd` mit den Nutzeranwendungen zu finden.

Die Speichermap weist verfügbaren Speicher aus. Nicht eingerechnet sind dabei die geladenen Module und auch nicht der Kernel. Diese müssen somit extra behandelt werden. Weiter sollte kein Speicher unter der 1 MiB Adressgrenze benutzt werden, da dort viele (vor allem ältere) Geräte eingeblendet sind (there will be the dragons!). Außerdem ist zwischen 15 MiB und 16 MiB das sogenannnte ISA-Hole, wo bei manchen älteren Mainboards noch diverse ISA Geräte eingeblendet sein können.

Weiterhin gilt:

- Der Multiboot-Header liegt an der Stelle im Speicher, auf die das Register `ebx` beim Kernelstart zeigt.
- Relevant sind nur die Informationen über den Speicher und die `initrd`.
- Die Einträge in der Speicherliste sind ungeordnet und widersprechen sich. Insbesondere können Bereiche, die initial als „frei und vorhanden“ ausgewiesen werden, später wieder als „belegt“ markiert werden. Ihr müsst die Einträge also immer konservativ behandeln (die „Belegt“-Markierung hat Vorrang)!
- Wir haben euch bereits verschiedene Datenstrukturen (im Namespace `Multiboot`) bereitgestellt, die den manchmal etwas seltsam anmutenden Multiboot-Header abstrahieren und bereitstellen. Diese werden beim Booten bereits korrekt initialisiert (Wie passiert das? Wo ist der Übertrag vom Register `ebx`?).

3.3 Virtuelles Layout

In StuBSmI verwenden wir die niedrigsten 32 MiB als Kernelspeicher und alles andere für Anwendungen. Der Kernelspace wird 1:1 gemappt und wird in alle Anwendungen eingeblendet (Meltdown lässt grüßen!). Wir können die Page Tables dafür recyclen.

Für die Erstellung neuer Objekte zur Laufzeit muss freier Speicher gefunden werden. In den niedrigen 32 MiB muss eine Halde für Kernelspeicher erstellt werden, für Anwendungen muss dann in den restlichen Speicherbereichen freier physischer Speicher gefunden werden, damit der seitenweise in den Adressraum der Prozesse gemappt werden kann.

Zwischenfragen

- Warum ist ein 1:1-Mapping praktisch (wenn die CPU keine unerwarteten Seitenkanäle hat)?
- Welche Page Tables können recycled werden?

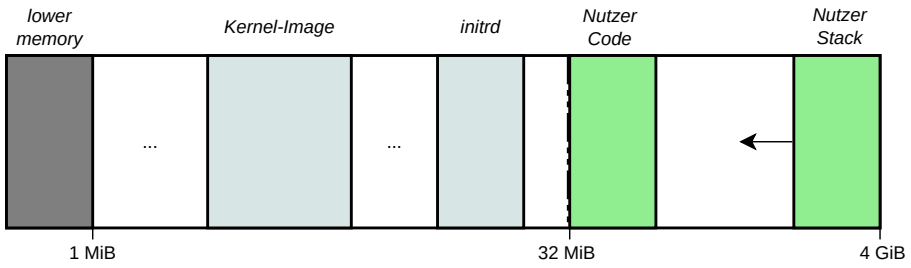


Abbildung 4: Speichermap für StuBSmI

dunkelgrau	there be the dragons
hellgrau	Kernel-space
grün	User-space

3.4 Kernelspeicher ($< 32\,\mathrm{MiB}$)

Für den Kern müssen wir eine Freispeicherliste pflegen, da der Kernel ab jetzt dynamisch allozierten Speicher benutzt. Für den Kernel muss daher gespeichert werden, wo in den unteren 32 MiB Adressraum freier und benutzbarer Speicher liegt. Eine Bitmap (bspw. über 4 KiB-Bereiche) wäre denkbar. Schneller und weniger speicherverschwendend wäre eine In-Place Freispeicherliste, bei der immer an den Anfang einer Page ein Pointer auf die nächste freie Page gelegt wird. Dann muss der Kernel sich lediglich einen `head`-Pointer speichern, der auf die erste Freispeicher-Node zeigt. **Achtung:** Code oder Daten des Kerns dürfen dabei keinesfalls überschrieben werden.

Zwischenfrage

- Welche Datenstrukturen sollten in dynamisch alloziertem Kernelspeicher liegen?
- Die verkettete Liste ist eine Möglichkeit für die Kernelfreispeicherliste, aber nicht für die Userfreispeicherliste. Warum?

3.5 Anwendungsspeicher ($\geq 32\,\mathrm{MiB}$)

Die Freispeicherliste aus dem Kern kann nicht einfach auf Anwendungsspeicher übertragen werden, weil der im Zweifel nicht gemappt ist, wenn eine Anfrage nach Speicherseiten kommt. Die einfachste Möglichkeit ist erneut eine Bitmap, die aber relativ groß ist (wie groß?).

Der Nachteil einer Bitmap ist vor allem die gleichbleibende statische Größe, da eine Bitmap immer den vollen 32 bit-Adressraum abdecken muss (bei 64 bit ist eine Bitmap z. B. komplett unbrauchbar). Naheliegend ist daher, die Datenstruktur des Allokators mitwachsen zu lassen. Wir haben im Kern bereits eine mitwachsende Datenstruktur, nämlich das Page Directory. Dieses alloziert nur Page Tables, wenn sie benötigt werden (und schmeißt sie idealerweise auch wieder weg, wenn sie nicht mehr benötigt werden). Es ist möglich, einen Allokator zu implementieren, der auf Page Directories basiert und den „Mitwachsen“-Mechanismus ausnutzt.

Wir legen uns dazu ein Page Directory an (`user_alloc_pd`) und interpretieren es als Mapping zwischen einem „Füllstand“ und einer „freien physischen Seite“. Den aktuellen Füllstand – also den größten Index – speichern wir uns extra ab (`fuellstand`). Dieser beschreibt die Anzahl der gemappten Seiten des Page Directories. Die ganze Datenstruktur kann somit als mitwachsende Liste interpretiert werden.

Die Operationen zum Frame anfordern (`get_frame`) und Frame freigeben (`free_frame`) sehen dann ungefähr so aus:

```
Frame get_frame() {
        Frame f = user_alloc_pd[fuellstand];
        --fuellstand;
        return f;
}

free_frame(Frame f) {
        ++fuellstand;
        user_alloc_pd[fuellstand] = f;
}
```

Dieser Allokator baut direkt auf dem Kernallokator auf, da Page Tables zur Laufzeit erzeugt (und damit beim Kernelallokator alloziert) werden, wenn sie benötigt werden, muss daher also zwangsläufig nach diesem initialisiert werden.

4 Bootvorgang in a nutshell

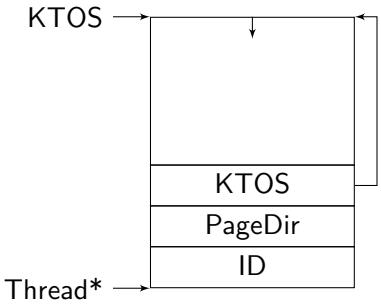


Abbildung 5: Threadstruktur in einer Page

1. Multiboot Header auslesen.
2. Allokatoren mit Speicher füllen (erst Kernel, dann User).
3. Anwendungen aus `initrd` anlegen, d.h.:
   a) Prozess- / Threadkontrollblöcke anlegen.
   b) Kernel-Stack anlegen (kann in einer Page passieren, siehe Abbildung 5).
   c) Page Directory anlegen (Kernel einblenden, User-Stack ans obere Ende des Adressraums blenden, Anwendungscode ans Ende des Kernel-Space legen).
   d) In `CR3` Page Directory setzen.
   e) In `CR0` Paging aktivieren.

Die Anwendungen beginnen ab dieser Aufgabe an einer genau definierten Adresse, d.h. Kickoff springt nicht länger zu `Thread::action`, sondern an die genau bestimmte virtuelle Adresse. Der User-Stack zeigt zu Beginn auf 0x00 und wächst von oben nach unten, sodass beim nächsten `push` der Wert ans obere Ende des Adressraums geschrieben wird.

5 Nutzeranwendungen

Die Anwendungen werden als so genanntes initiales RAM-FS vom Bootloader geladen, also nicht mehr länger als Teil unseres Betriebssystems ausgeliefert. Sie müssen somit separat kompiliert werden. Nutzeranwendungen und Kerncode sollten also abgetrennt sein, um unabhängig voneinander gebaut werden zu können.

Die Anwendungen müssen nach dem Bauen „flachgeklopft“ werden, d. h., dass die BSS- und Datensegmente aus der gebauten ELF-Datei als Teil der Images von Nutzeranwendungen integriert werden sollen. Weiter ist es wichtig, die Konstruktoren aller Objekte zu rufen (das passiert normalerweise über den Initilialisierungscode des Compilers, der muss jetzt aber selbst explizit bereitgestellt werden).

- Bauen aller C++-Dateien zu Objektdateien, ohne Benutzung der Standardbibliotheken
  `g++ -m32 -fomit-frame-pointer -ffreestanding -fno-builtin -nodefaultlibs -nostdlib -nostdinc -fno-tree-loop-distribute-patterns -nostartfiles -mno-mmx -mno-sse -Wno-non-virtual-dtor -fno-rtti -fno-exceptions -Wno-write-strings -fno-stack-protector -mno-red-zone <files>`

- Linken der Objektdateien, zusammen mit der Syscall-Bibliothek für StuBSmI und der `init.cc`, die eine `init`-Funktion enthält. Diese muss am Anfang des Binaries stehen, damit die Konstrukturen gerufen werden.
  `ld -T user.sections.ld -o app -melf_i386 <lib32/crti.o, crtbegin.o, crtend.o, crtn.o>`[5]
- Flachklopfen der ELF-Dateien zu einem Image mit `objcopy`
  `objcopy -O binary --set-section-flags .bss=alloc,load,contents app app.img`[6]
- Zusammenführen aller Anwendungsimages zu einem `initrd` (mittels `imgbuilder`)
  `./imgbuilder app.img app2.img > initrd.img`

Der imgbuilder richtet alle Anwendungen auf Pages aus. Die `initrd` hat das in Abbildung 6 dargestellte Format.

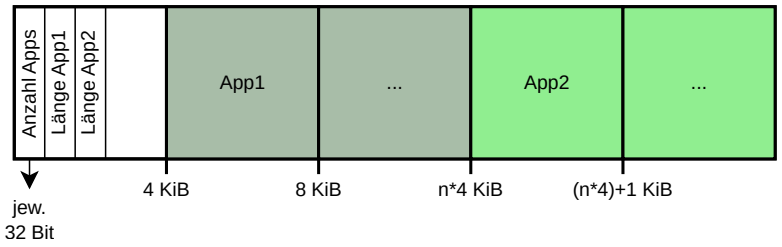


Abbildung 6: Layout der `initrd`

6 Hinweise

- **Benutzt ordentliche C++-Abstraktionen für Page Directories und Page Tables!**
- Das `kickoff` springt jetzt direkt an das Ende der Kernspeichers (Warum?).
- Der Userstackpointer ist bei 0x00 (Warum?).

[5] Aus Platzgründen abgekürzt
[6] Eingabe app, Ausgabe app.img

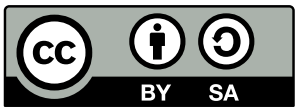
Literatur

[1] Multiboot Specification version 0.6.96. `https://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format`. [zugegriffen 22.10.2019].

[2] Paging - OSDev Wiki. `https://wiki.osdev.org/Paging`. [zugegriffen 17.10.2019].

[3] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3*, 2017. `https://www.sra.uni-hannover.de/Lehre/SS19/V_BSB/doc/intel_manual_2017-03.pdf`.

Lizenz



Es wurde von Stefan Naumann, Christian Dietrich und Gerion Entrup erstellt.