Tutorübung Grundlagen: Betriebssysteme und Systemsoftware

Moritz Beckel

München, 1. Februar 2023

Mittwoch 14:15-16:00 Uhr Online (https://bbb.in.tum.de/mor-6ij-iuw-ypm)

Zulip-Stream https://zulip.in.tum.de/#narrow/stream/1296-GBS-Mi-1400-A

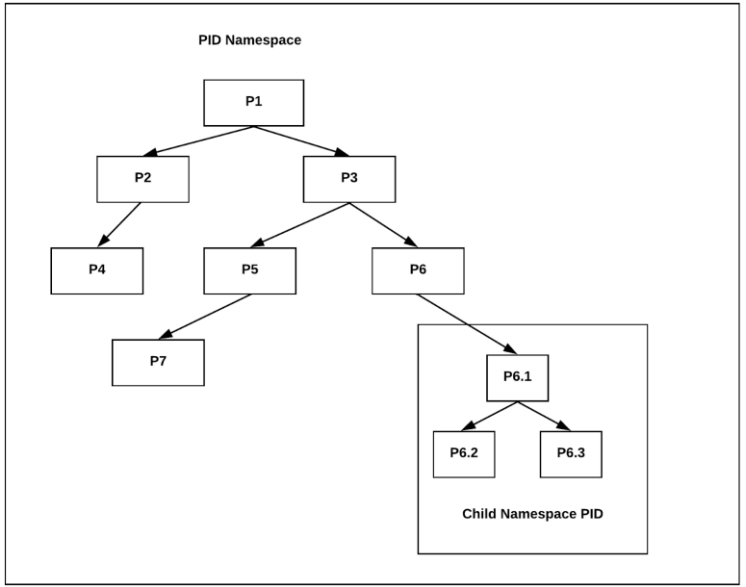
Unterrichtsmaterialien findest du hier:

https://home.in.tum.de/~beckel/gbs

Folien wurden von mir selbst erstellt. Es besteht keine Garantie auf Korrektheit.

Linux Namespaces

1. Mechanismus zum Isolieren bestimmter Ressourcen
2. Prozesse können nur eine Untermenge an Ressourcen sehen (bspw. PIDs, Hostnames, UIDs)
3. Wird von Containern unter anderem zur Isolation benutzt (bspw. Docker)



Aufgabe 2

a) Welche Arten von Linux-Namespaces gibt es? Was wird über diese jeweils isoliert?
(Hinweis: Die Manpage *man 7 namespaces* liefert Auskunft)

Linux Namespaces

1. User-Namespaces: Isolieren die Nutzer- und Gruppen-IDs auf einem System und erlauben separate Vergaben von Rechten (sog. Capabilities).
2. Mount-Namespaces: Isolieren die Mountpoints im virtual file system (VFS)
3. PID-Namespaces: Isolieren die Prozesse und PIDs
4. Network-Namespaces: Isolieren die Netzwerkstacks (Interfaces, Adressen, Firewall etc.)
5. UTS-Namespaces: (UTS = Unix Time Share) Isolieren Host- und Domainname des Hosts

Für uns weniger relevant: Time-Namespaces, IPC-Namespaces

Aufgabe 2

b) Wechseln Sie mithilfe des unshare-Befehls in einen neuen User-Namespace. Welche Auswirkungen bemerken Sie? Erstellen Sie innerhalb des Namespaces eine neue Datei und vergleichen Sie die Dateiattribute innerhalb und außerhalb des Namespaces.

```
$ unshare --user --map-root-user
```

Aufgabe 2

b) Wechseln Sie mithilfe des unshare-Befehls in einen neuen User-Namespace. Welche Auswirkungen bemerken Sie? Erstellen Sie innerhalb des Namespaces eine neue Datei und vergleichen Sie die Dateiattribute innerhalb und außerhalb des Namespaces.

- Durch --map-root-user werden alle Operationen, die wir innerhalb des Namespaces mit der UID 0 (root) ausführen, außen mit den Rechten des Nutzers durchgeführt, der den unshare-Systemcall aufgerufen hat. Während die neue Datei im Namespace also nun root gehört, gehört sie außen nach wie vor unserem regulären Nutzer.

- Dieses Mapping geht natürlich auch in die andere Richtung: Sobald wir im Namespace z.B. mit ls -l abfragen, welchem Nutzer eine Datei gehört, so muss der Linux-Kernel erst ermitteln, welcher Nutzer im Namespace auf den Besitzer der Datei außen mapt.

Aufgabe 2

c) Erstellen Sie nun mit unshare einen neuen mount-Namespace und hängen Sie über den mount-Befehl ein temporäres Dateisystem (tmpfs) in einem Verzeichnis Ihrer Wahl ein. Erstellen Sie im eingehängten Dateisystem wieder ein paar Dateien und untersuchen Sie die Unterschiede zwischen der Ansicht innerhalb und außerhalb des Namespaces.

```
$ unshare --mount --user --map-root-user
$ mkdir ./mountpoint
$ mount -t tmpfs none ./mountpoint
```

Aufgabe 2

c) Erstellen Sie nun mit unshare einen neuen mount-Namespace und hängen Sie über den mount-Befehl ein temporäres Dateisystem (tmpfs) in einem Verzeichnis Ihrer Wahl ein. Erstellen Sie im eingehängten Dateisystem wieder ein paar Dateien und untersuchen Sie die Unterschiede zwischen der Ansicht innerhalb und außerhalb des Namespaces.

- Dadurch, dass wir getrennte Mount-Namespaces haben, können wir innerhalb und außerhalb des Namespaces getrennte Mountpoints haben. Insbesondere hat ein neuer Mount innerhalb des Namespaces keinerlei Auswirkungen außerhalb des Namespaces. Daher haben wir nun unter ./mountpoint innerhalb des Namespaces ein neues tmpfs, in welches wir beliebig Dateien legen können, ohne dass sie außerhalb des Namespaces erscheinen.

Aufgabe 2

d) Als nächstes wollen wir nun PID-Namespaces betrachten. Wechseln Sie mit unshare in einen neunen PID namespace. In der Kommandozeile können Sie über die $$-Variable die PID des aktuellen Shell-Prozesses ausgeben lassen. Was erwarten Sie wird passieren, wenn Sie sich innerhalb des Namespaces die aktuell laufenden Prozesse in einem Process-Viewer wie htop oder ps anschauen? Erklären Sie Ihre Beobachtungen!

```
$ unshare --pid --fork --user --map-root-user
$ echo $$
$ ps -aux | less
```

Aufgabe 2

d) Als nächstes wollen wir nun PID-Namespaces betrachten. Wechseln Sie mit unshare in einen neunen PID namespace. In der Kommandozeile können Sie über die $$-Variable die PID des aktuellen Shell-Prozesses ausgeben lassen. Was erwarten Sie wird passieren, wenn Sie sich innerhalb des Namespaces die aktuell laufenden Prozesse in einem Process-Viewer wie htop oder ps anschauen? Erklären Sie Ihre Beobachtungen!

- Unsere Shell hat innerhalb des Namespaces die PID 1. In der Prozess-Liste hingegen ist Prozess 1 aber nach wie vor der init-Prozess. Der Grund dafür ist, dass ps, htop etc. für die Prozessliste in /proc schauen, welches innerhalb und außerhalb des Namespaces identisch ist. Damit diese Programme die erwartete Liste an Prozessen innerhalb des Namespaces ausgeben, müssen wir /proc neu mounten: Dafür ergänzen wir beim unshare-Befehl den Parameter --mount und führen direkt danach mount proc /proc -t proc aus.

Aufgabe 2

e) Inwiefern kann es geschickter sein, Linux Namespaces statt regulären virtuellen Maschinen zu verwenden?

Aufgabe 2

e) Inwiefern kann es geschickter sein, Linux Namespaces statt regulären virtuellen Maschinen zu verwenden?

- In regulären VMs wird der gesamte Rechner inkl. Prozessor, Speicher, Peripheriegeräten und Betriebssystemen virtualisiert. Damit geht ein großer Overhead und ggf. auch ein Leistungsverlust einher. Container, welche auf Linux Namespaces basieren, sind in erster Linie nur Prozesse auf dem selben Betriebssystem, hier gibt es keinerlei Performance-Einbußen.

Aufgabe 2

f) (optional) Betrachten Sie den folgenden C-Code. Wo werden die Namespaces erstellt? Wann bekommt der Prozess die PID 1? Welches Mapping der UIDs und GIDs wird erstellt? Was macht das mount in der Zeile 43?

Aufgabe 2

f) (optional) Betrachten Sie den folgenden C-Code. Wo werden die Namespaces erstellt? Wann bekommt der Prozess die PID 1? Welches Mapping der UIDs und GIDs wird erstellt? Was macht das mount in der Zeile 43?

- Die Namespaces werden mit dem unshare-Systemcall in der Zeile 14 erstellt. Der Kindprozess ist derjenige, welcher am Ende die PID 1 haben wird. Diese erhält er mit dem fork in der Zeile 16.
- Das Mapping für die UID wird in den Zeilen 26 bis 29 festgelegt, indem es in die Datei /proc/self/uid_map geschrieben wird. Dabei wird eine (1) UID festgelegt, welche von root im Namespace (0) auf die UID des Nutzers außen (uid) mappen soll.
- Das mount in der Zeile 43 mountet das Proc-FS neu, damit die Liste an Prozessen, die durch ps abgefragt werden kann, an das neue PID-Namespace angepasst ist.

Aufgabe 2

g) Versuchen Sie selbstständig, mittels des unshare-Befehls noch einige der verbleiben Namespace-Typen auszuprobieren.

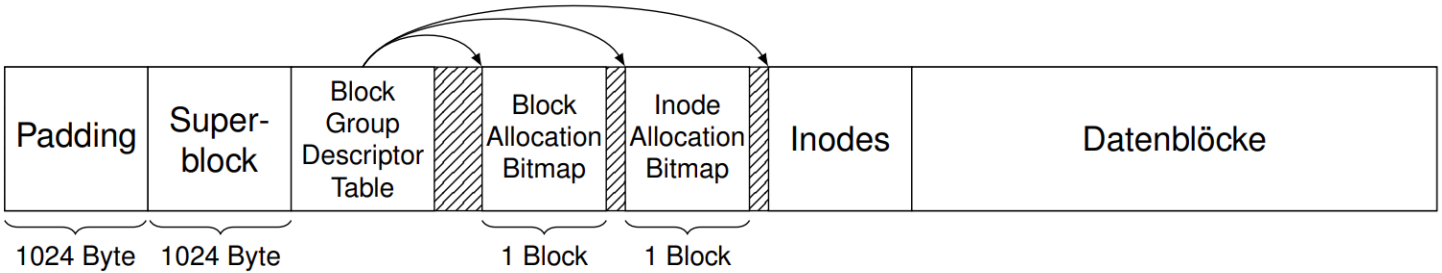
Aufgabe 2

g) Versuchen Sie selbstständig, mittels des unshare-Befehls noch einige der verbleiben Namespace-Typen auszuprobieren.

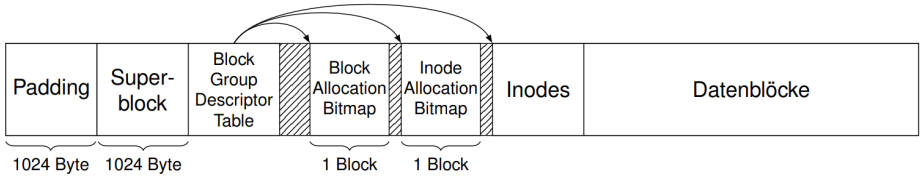
- Mithilfe des UTS-Namespace (mit *--uts*) lässt sich der Hostname des Rechners ändern über dem Befehl *hostname.*
- Mithilfe des Network-Namespace (mit --net) kann man sich ein virtuelles Network-Stack erstellen auf dem wir Netzwerk-Geräte erstellen und Adressen vergeben können.

Aufgabe 3

Die Familie der ext{2,3,4}-Dateisysteme unterteilt das Dateisystem in mehrere Blöcke zwischen 1 KiB und 64 KiB Größe. Diese werden in mehrere Block Groups zusammengefasst, wovon jede Platz für eine feste Anzahl an Inodes und Datenblöcken bietet. Eine Block Group hat im Groben folgendes Layout:



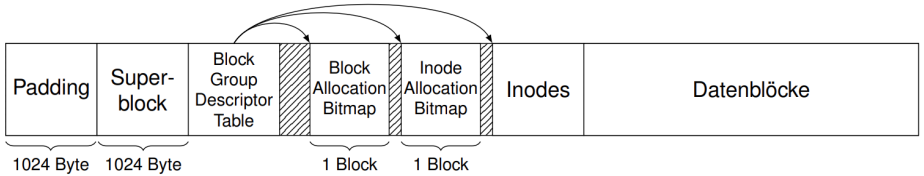
Aufgabe 3



a) Finden Sie im Hexdump (hd ext_image.img | less) den Superblock.
(zum runterladen: wget https://gbs.cm.in.tum.de/media/material/ext_image.img)

- Wie groß ist ein Block (1 << (10 + s_log_block_size))?
- Wie viele Block Groups hat das Dateisystem (s_inodes_count / s_inodes_per_group)?
- Wie groß ist jede Inode (s_inode_size)?

Aufgabe 3

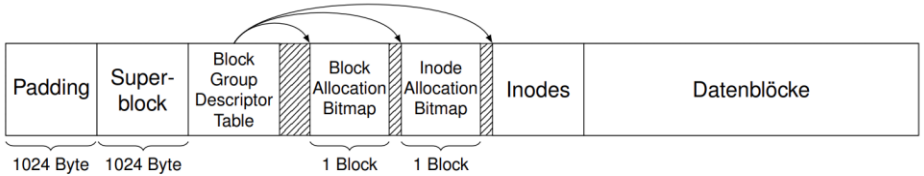


a) Finden Sie im Hexdump (hd ext_image.img | less) den Superblock.
(zum runterladen: wget https://gbs.cm.in.tum.de/media/material/ext_image.img)

- Wie groß ist ein Block (1 << (10 + s_log_block_size))?
- Wie viele Block Groups hat das Dateisystem (s_inodes_count / s_inodes_per_group)?
- Wie groß ist jede Inode (s_inode_size)?

- 2^{10} B = 1 KiB
- 0x80 / 0x80 = 1 Block Group
- 0x100 B = 256 B

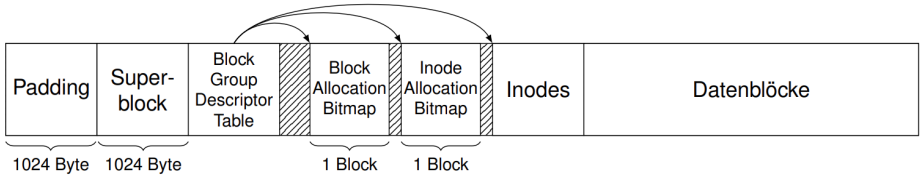
Aufgabe 3



b) Finden Sie im Hexdump die Block Group Descriptor Table. Jeder Eintrag ist 64 Byte groß. Bestimmen Sie für die erste Block Group die Adressen der:
(Blockgröße = 1024 B = 0x400 B)

- Block Allocation Bitmap (bg_block_bitmap_lo · Blockgröße)
- Inode Allocation Bitmap (bg_inode_bitmap_lo · Blockgröße)
- Inodes (bg_inode_table_lo · Blockgröße)

Aufgabe 3

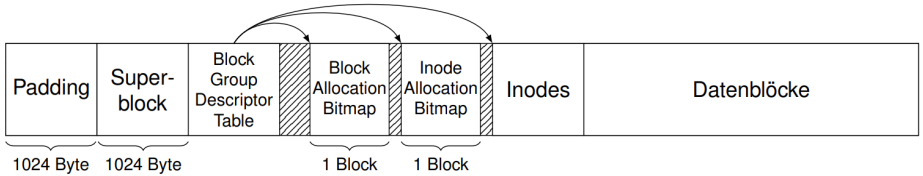


b) Finden Sie im Hexdump die Block Group Descriptor Table. Jeder Eintrag ist 64 Byte groß. Bestimmen Sie für die erste Block Group die Adressen der:

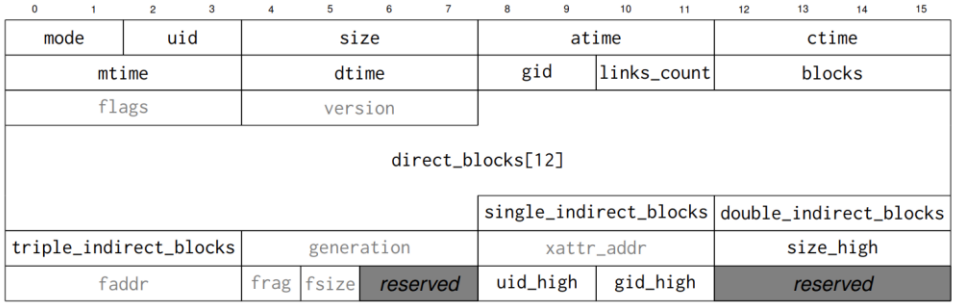
- Block Allocation Bitmap (bg_block_bitmap_lo · Blockgröße)
- Inode Allocation Bitmap (bg_inode_bitmap_lo · Blockgröße)
- Inodes (bg_inode_table_lo · Blockgröße)

- 6 · 0x400 = 0x1800
- 7 · 0x400 = 0x1c00
- 8 · 0x400 = 0x2000

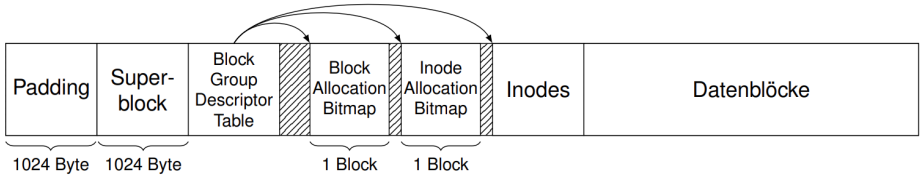
Aufgabe 3



c) Nehmen Sie im Folgenden an, dass alle Inodes in unserem Image gleich groß sind (s_inode_size im Superblock, 256 B). Finden Sie die Inode für das Root-Directory (Inode 2). Beachten Sie, dass die Inode-Nummerierung mit 1 beginnt. Wem gehört das Verzeichnis? Welche Rechte sind vergeben? (Inode-Tabelle beginnt bei 0x2000)



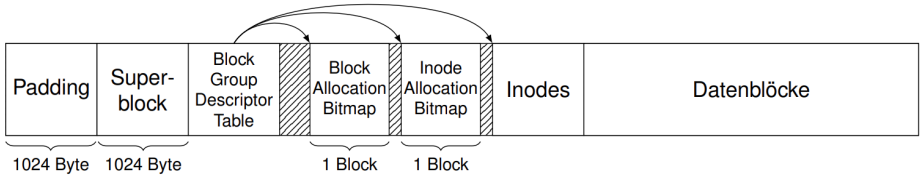
Aufgabe 3



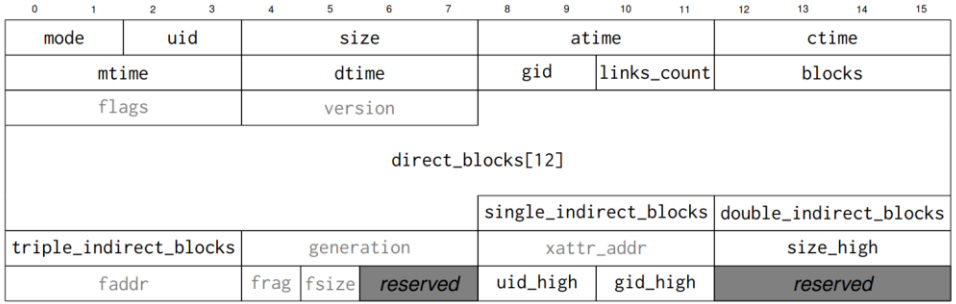
c) Nehmen Sie im Folgenden an, dass alle Inodes in unserem Image gleich groß sind (s_inode_size im Superblock, 256 B). Finden Sie die Inode für das Root-Directory (Inode 2). Beachten Sie, dass die Inode-Nummerierung mit 1 beginnt. Wem gehört das Verzeichnis? Welche Rechte sind vergeben? (Inode-Tabelle beginnt bei 0x2000)

- Jede Inode ist 0x100 = 256 Byte groß. Die Inode 2 liegt daher an der Adresse $0x2000 + (2 - 1) \cdot 0x100 = 0x2100$.
- Die UID beginnt laut dem Tutorblatt der letzen Woche an Byte 2 in der Inode. Wir lesen also die Nutzer-ID als 16-Bit little-endian Wert aus der Adresse 0x2102: 0 ⇒ root.
- Die Rechte sind die unteren 9 Bit des modes, welche wir direkt an der Adresse 0x2100 als 16-Bit little-endian Wert auslesen können: $1ed_{16} = 755_8$ ⇒ rwxr-xr-x

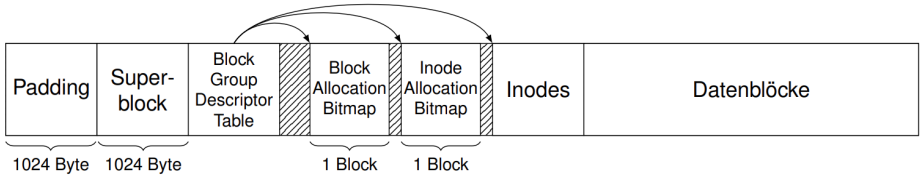
Aufgabe 3



d) (optional) Lesen Sie die Einträge des Root-Directories aus. Beachten sie den Hinweis in der Teilaufgabe e). (Adresse der Inode: 0x2100, Blockgröße 0x400)



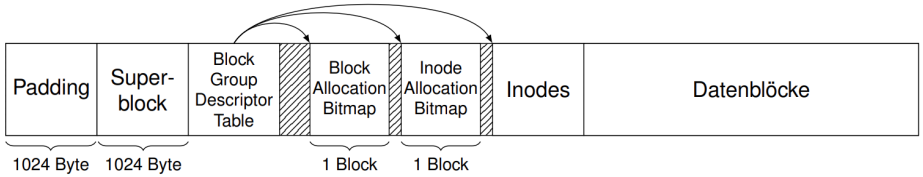
Aufgabe 3



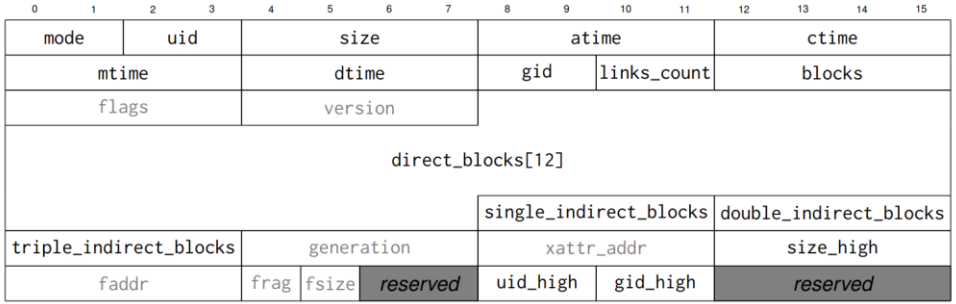
d) (optional) Lesen Sie die Einträge des Root-Directories aus. Beachten sie den Hinweis in der Teilaufgabe e). (Adresse der Inode: 0x2100, Blockgröße 0x400)

- Laut der Inode ist das Root-Directory 1 KiB groß, benötigt also nur einen Block: 0x28. Dieser Block liegt an der Adresse $0x28 \cdot 0x400 = 0xa000$.

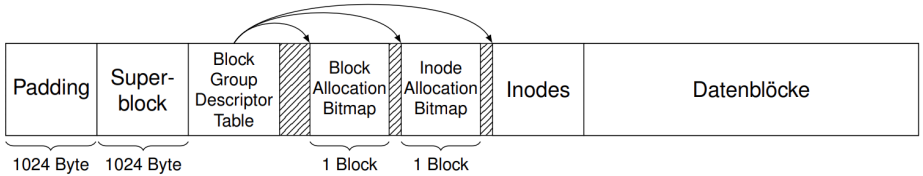
Aufgabe 3



e) Wir betrachten nun die Datei file.txt (vorherige Teilaufgabe ergibt: Inode Nummer 11). Lesen Sie den Dateiinhalt aus. (Basisadresse: 0x2000, Inode-Größe 0x100, Blockgröße 0x400)



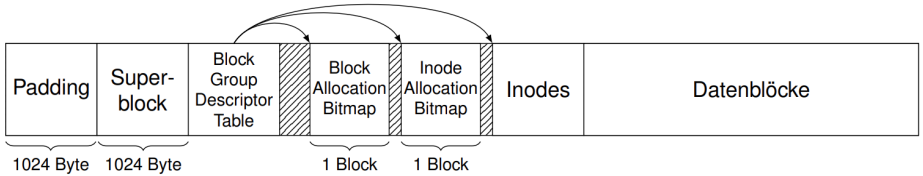
Aufgabe 3



e) Wir betrachten nun die Datei file.txt (vorherige Teilaufgabe ergibt: Inode Nummer 11). Lesen Sie den Dateiinhalt aus. (Basisadresse: 0x2000, Inode-Größe 0x100, Blockgröße 0x400)

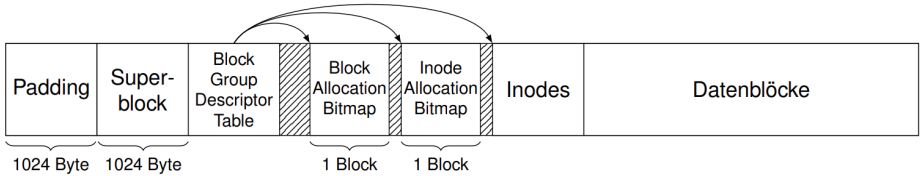
- Wir finden die Inode an der Adresse $0x2000 + (11 - 1) \cdot 0x100 = 0x2a00$. Laut der Inode ist die Datei 0xe = 14 Byte groß.
- Wir belegen damit nur einen (direct) Block, welchen wir aus der Adresse 0x2a28 ermitteln: 0x29.
- Dieser liegt also an der Adresse $0x29 \cdot 0x400 = 0xa400$. Dort lesen wir 14 Byte aus: "GBS ist toll!\n"

Aufgabe 3



f) (optional) Betrachten Sie die Bitmaps. Können Sie damit eine gelöschte Inode wieder finden? Rekonstruieren Sie die Datei!
Hinweis: Die Bitreihenfolge der Bitmaps ist auf den ersten Blick unintuitiv: In jedem Byte steht das niederwertige Bit für den ersten, das höchstwertige Bit für den letzten Block.
(Inode Bitmap an Adresse 0x1c00, Basisadresse Inodes 0x2000)

Aufgabe 3



f) (optional) Betrachten Sie die Bitmaps. Können Sie damit eine gelöschte Inode wieder finden? Rekonstruieren Sie die Datei!
Hinweis: Die Bitreihenfolge der Bitmaps ist auf den ersten Blick unintuitiv: In jedem Byte steht das niederwertige Bit für den ersten, das höchstwertige Bit für den letzten Block.
(Inode Bitmap an Adresse 0x1c00, Basisadresse Inodes 0x2000)

- Ein Blick in die Inode-Bitmap ergibt: Alle Inodes von 1 bis 17 sind belegt, bis auf Inode 13. Ein Blick in Inode 13 (Adresse 0x2c00) ergibt, dass hier einst eine Datei stand. Diese war 0x2b = 43 Byte groß und lag in Block 0x2b, also an Adresse 0xac00. Mit etwas Glück wurde dieser Block noch nicht überschrieben. Und tatsächlich, im Block stehen genau diese 43 Byte: "flag{2586d8ab8c8f13a37ffe1ccd4b5f7f6e788c}\n"