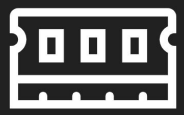
Ninjastorms

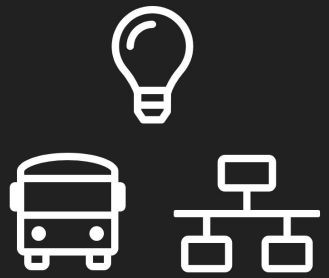


Memory Management

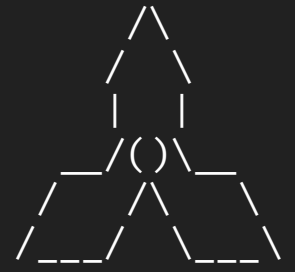
Marcel Garus · Rohan Sawahn · Jonas Wanke

Ninjastorms

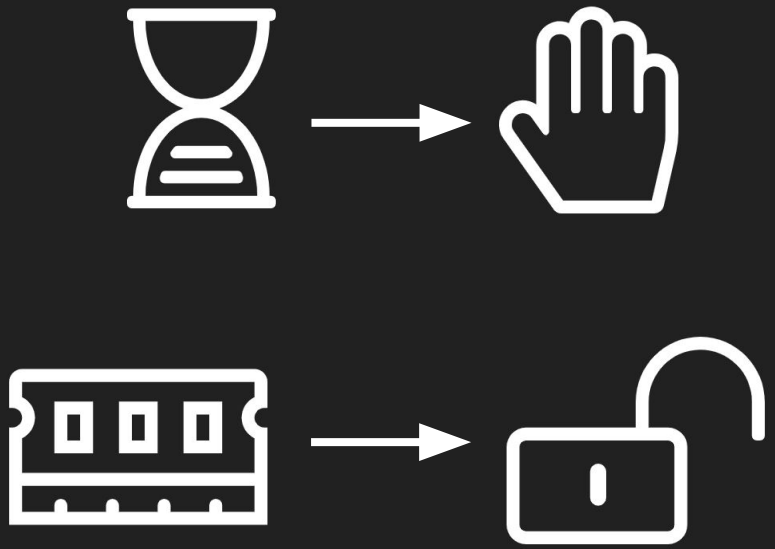
Ninjastorms



```
$ qemu-system-arm -M versatilepb -m 128M -nographic -kernel ninjastorms

This is ninjastorms OS
  shuriken ready
```



Ninjastorms: Tasks



Memory Management

Ziel: Zugriffe auf RAM verwalten

effizient

MMU-Hardware
Mapping

sicher

Berechtigungen
Adressraumisolation

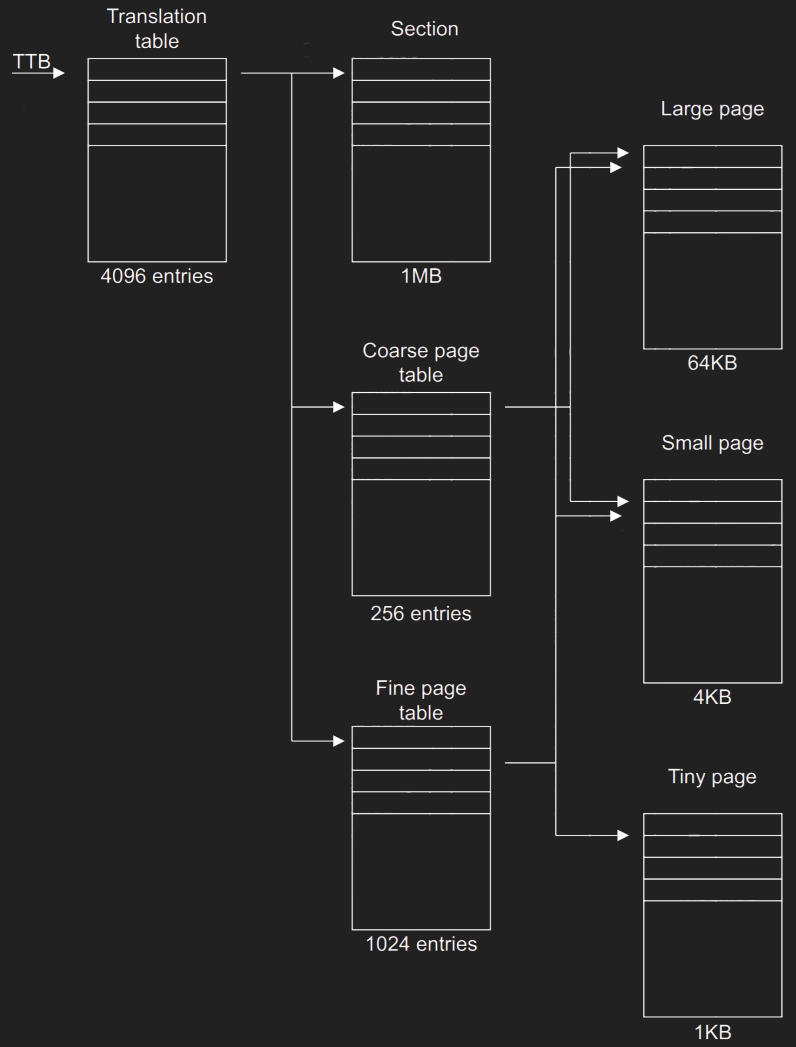
komfortabel

Demand Paging
Swapping

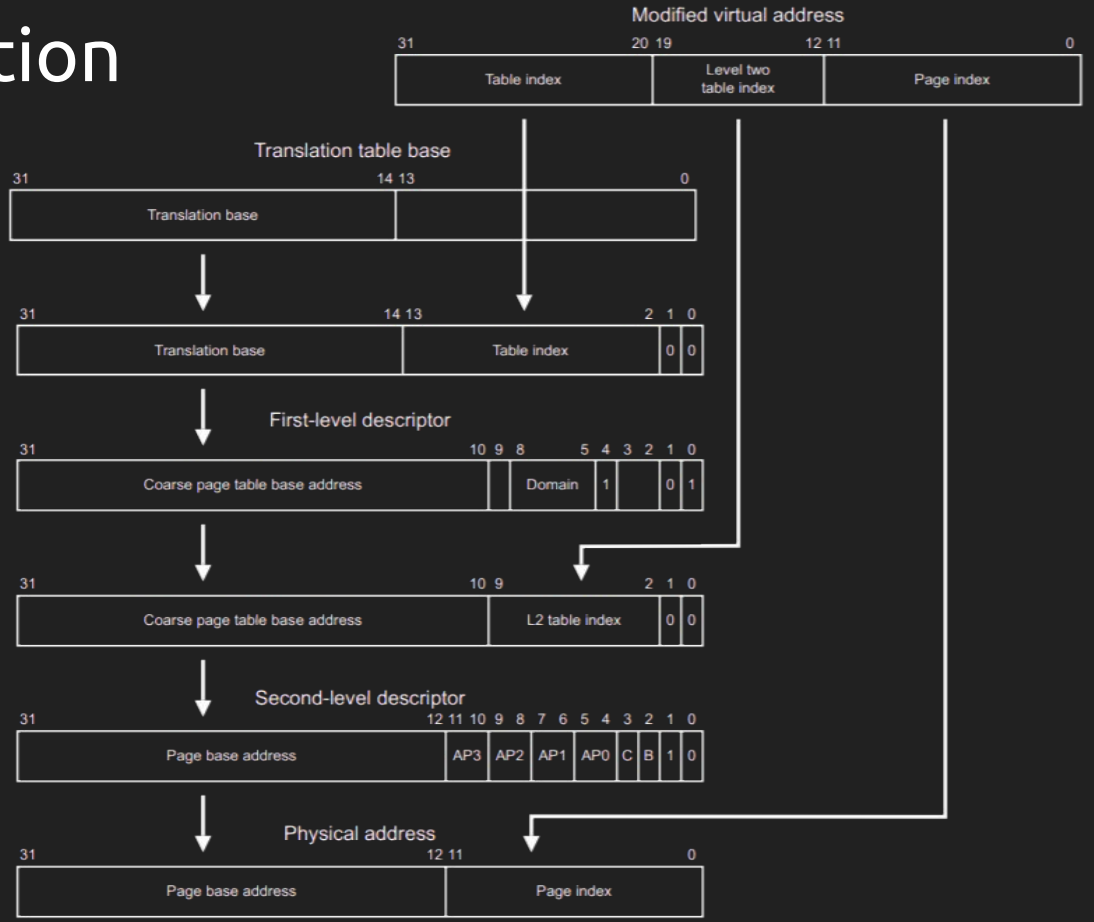
Hardware des Lego Mindstorms EV3

Prozessor	AM1808 · ARM
RAM	64 MB · DDR
Co-Prozessor inkl. MMU	ARM926EJ-S

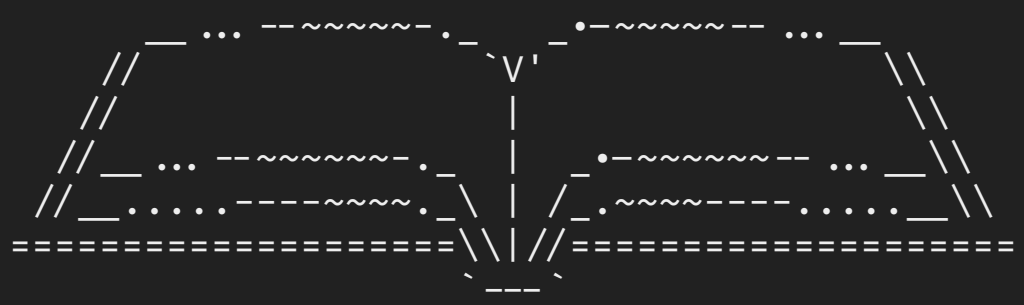
- Virtueller Adressraum: 4 GiB
- Automatischer TLB-Cache
- Erzeugt Page Faults



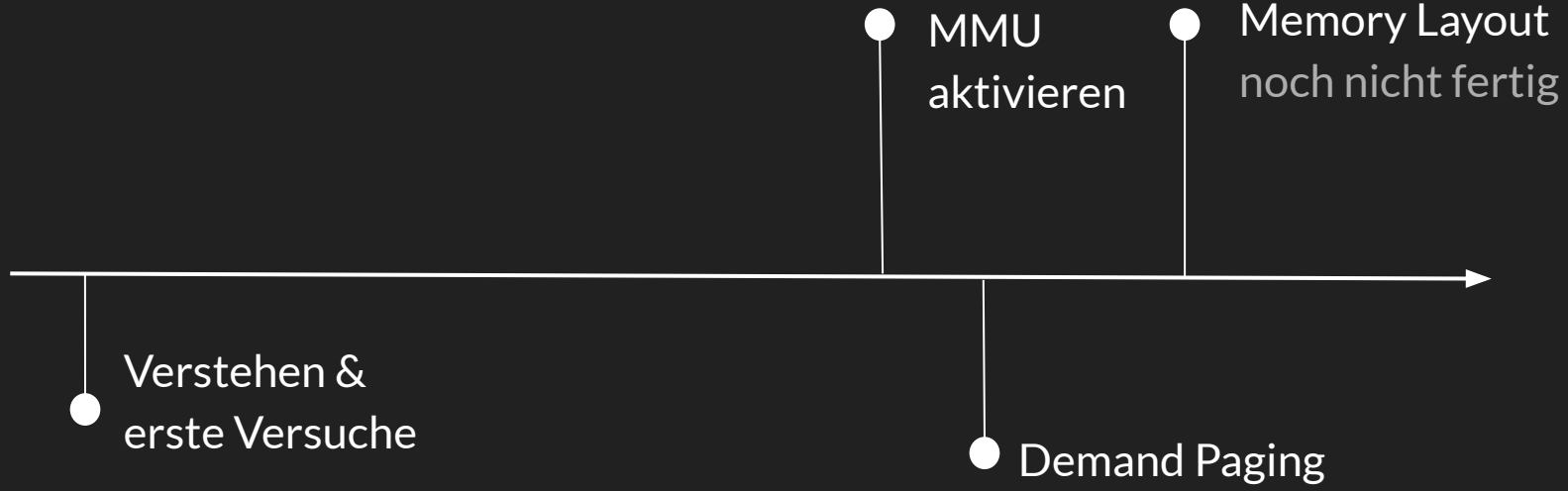
Page Translation



A short history…

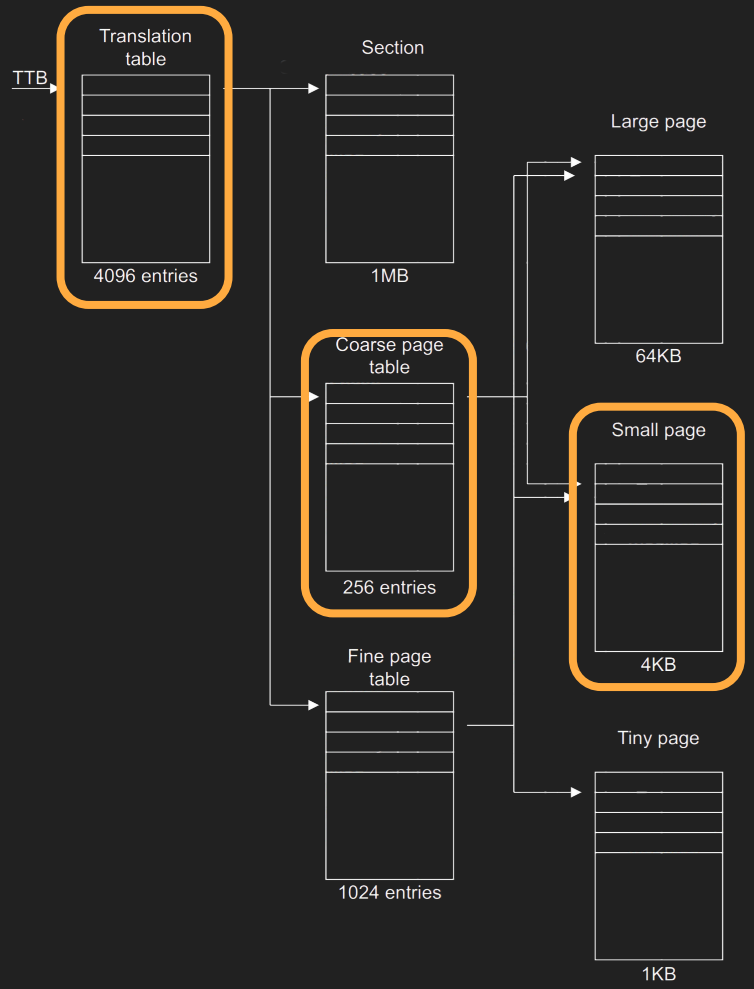


Grober Projektverlauf

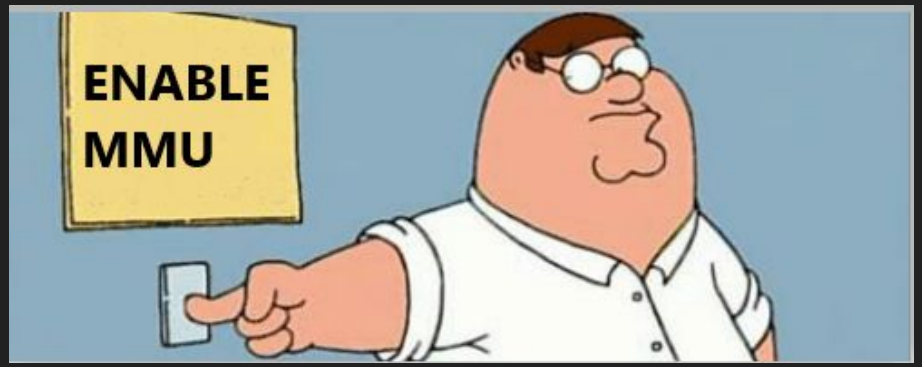


Verstehen & erste Versuche

- Bestehendes Ninjastorms OS · Prozessor · MMU · Kommunikation
- Entscheidung: Überall 2 Level Paging mit Coarse & Small Pages
  - Retrospektive: Erzeugt Overhead für fest gemappten Speicher



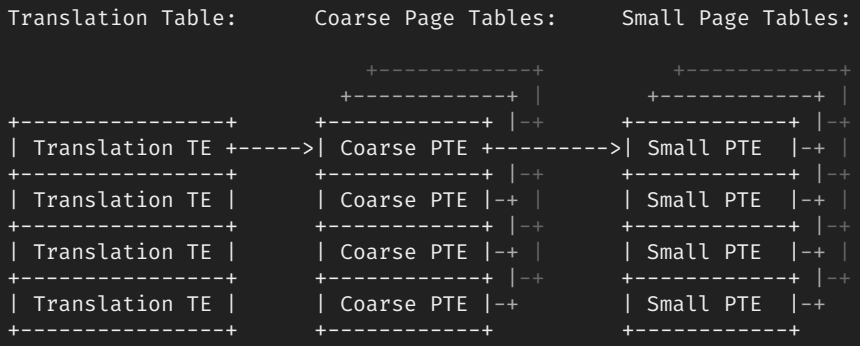
Paging aktivieren & Prozessor läuft noch

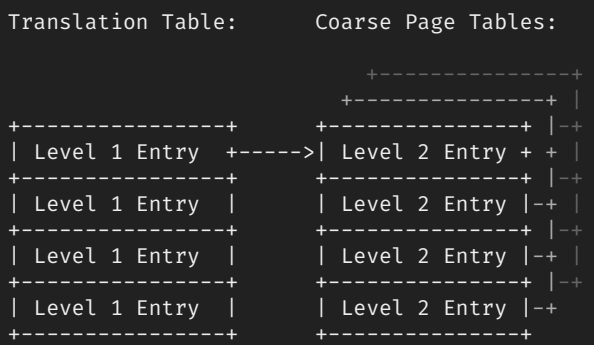


Dazu:
Identity-Mapping

```
// Actually enable the MMU.
asm (
  "MRC p15, 0, R1, c1, c0, 0\n" // Read control register
  "ORR R1, #0x1\n"              // Set M-bit from read Control Register to 1
  "MCR p15, 0, R1, c1, c0, 0\n" // Write control register and enables MMU
);
```

3- statt 2-Level-Paging implementiert





Bitpacking funktioniert nicht

```
// # Level 2 entry
// Memory layout:
// | 31        12 | 11 10 | 9 8 | 7 6 | 5 4 | 3 | 2 | 1      0 |
// | Base address |   AP3 | AP2 | AP1 | AP0 | C | B | Validity |
```

Mit Bitpacking

```
struct mem_lvl2_entry_t
{
  uint32_t base_address: 20;

  uint8_t ap3: 2;
  uint8_t ap2: 2;
  uint8_t ap1: 2;
  uint8_t ap0: 2;

  uint8_t c: 1;
  uint8_t b: 1;

  uint8_t descriptor: 2;
} __attribute__((packed));
```

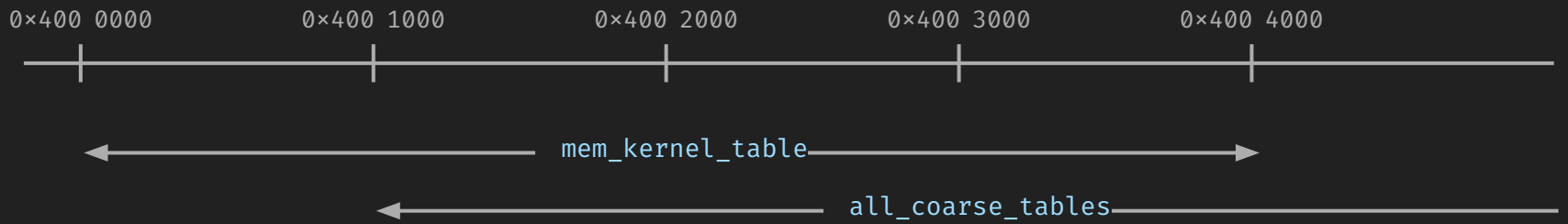
Ohne Bitpacking

```
#define LVL2_ENTRY_DEFAULT 0b00000000000000000000000000000000

#define LVL2_ENTRY_GET_BASE_ADDRESS(entry) ENTRY_GET(entry, 12, 20)
#define LVL2_ENTRY_GET_BASE_POINTER(entry) \
   ((void*) LVL2_ENTRY_GET_BASE_ADDRESS(entry) << 12)
#define LVL2_ENTRY_SET_BASE_ADDRESS(entry, value) \
   ENTRY_SET(entry, value, 12, 20)
#define LVL2_ENTRY_SET_BASE_POINTER(entry, value) \
   LVL2_ENTRY_SET_BASE_ADDRESS(entry, ((uint32_t) (value)) >> 12)

#define LVL2_ENTRY_GET_AP3(entry) ENTRY_GET(entry, 10, 2)
#define LVL2_ENTRY_SET_AP3(entry, value) ENTRY_SET(entry, value, 10, 2)
#define LVL2_ENTRY_GET_AP2(entry) ENTRY_GET(entry, 8, 2)
#define LVL2_ENTRY_SET_AP2(entry, value) ENTRY_SET(entry, value, 8, 2)
#define LVL2_ENTRY_GET_AP1(entry) ENTRY_GET(entry, 6, 2)
#define LVL2_ENTRY_SET_AP1(entry, value) ENTRY_SET(entry, value, 6, 2)
#define LVL2_ENTRY_GET_AP0(entry) ENTRY_GET(entry, 4, 2)
#define LVL2_ENTRY_SET_AP0(entry, value) ENTRY_SET(entry, value, 4, 2)
```

Überschriebene Tabellen

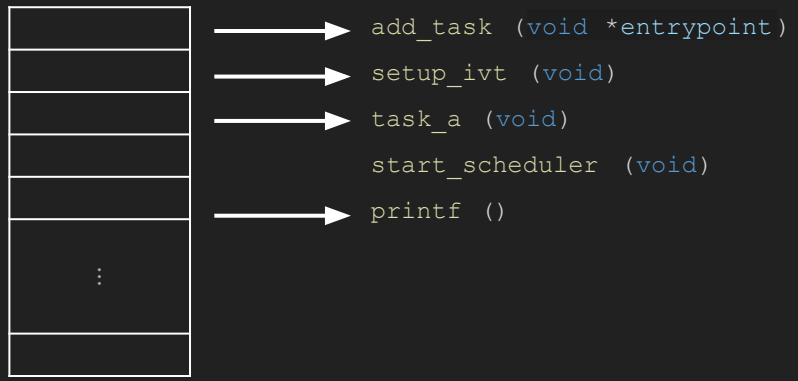
Ist `0×00 4000` groß

```
mem_translation_table_t *mem_kernel_table = (mem_translation_table_t *) 0×400 0000;
mem_coarse_table_t *all_coarse_tables = (mem_coarse_table_t *) 0×400 1000;
```



Zu wenige Tabellen initialisiert

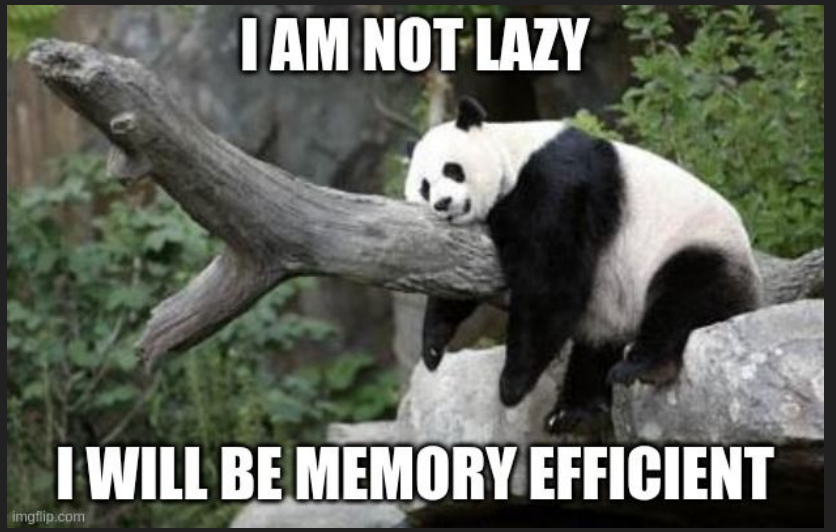
Alle benutzten Funktionen müssen gemapped werden.



Paging aktivieren & Prozessor läuft noch

Demand Paging

- Lazy Initialisierung von Pages
  - Mit minimaler Anzahl starten
  - Bei Data/Prefetch Abort neue Seite erstellen
- In Kombination mit Swapping sinnvoll



Interrupts werden nicht behandelt

- Prozessor hat verschiedene Modi je nach Interrupt-type
- Spezieller Abort Stack benötigt

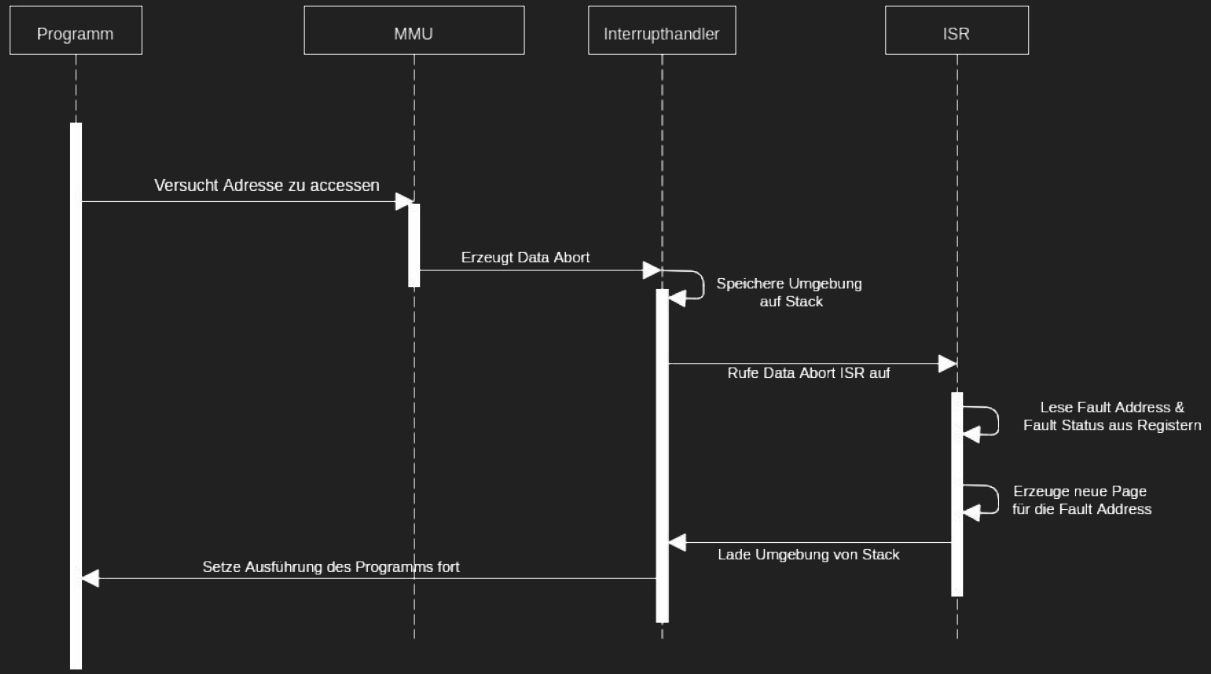
setup_irq_stack()

```
"orr  r0, #0x12  \n" // Select IRQ mode
"msr  cpsr, r0  \n"  // Enter IRQ mode
"mov  sp, %[irq_stack_address]  \n"  // set stack pointer
...
```

setup_abt_stack()

```
"orr  r0, #0x17 \n" // Select ABT mode
"msr  cpsr, r0  \n"  // Enter ABT mode
"mov  sp, %[abt_stack_address] \n"  // set stack pointer
...
```

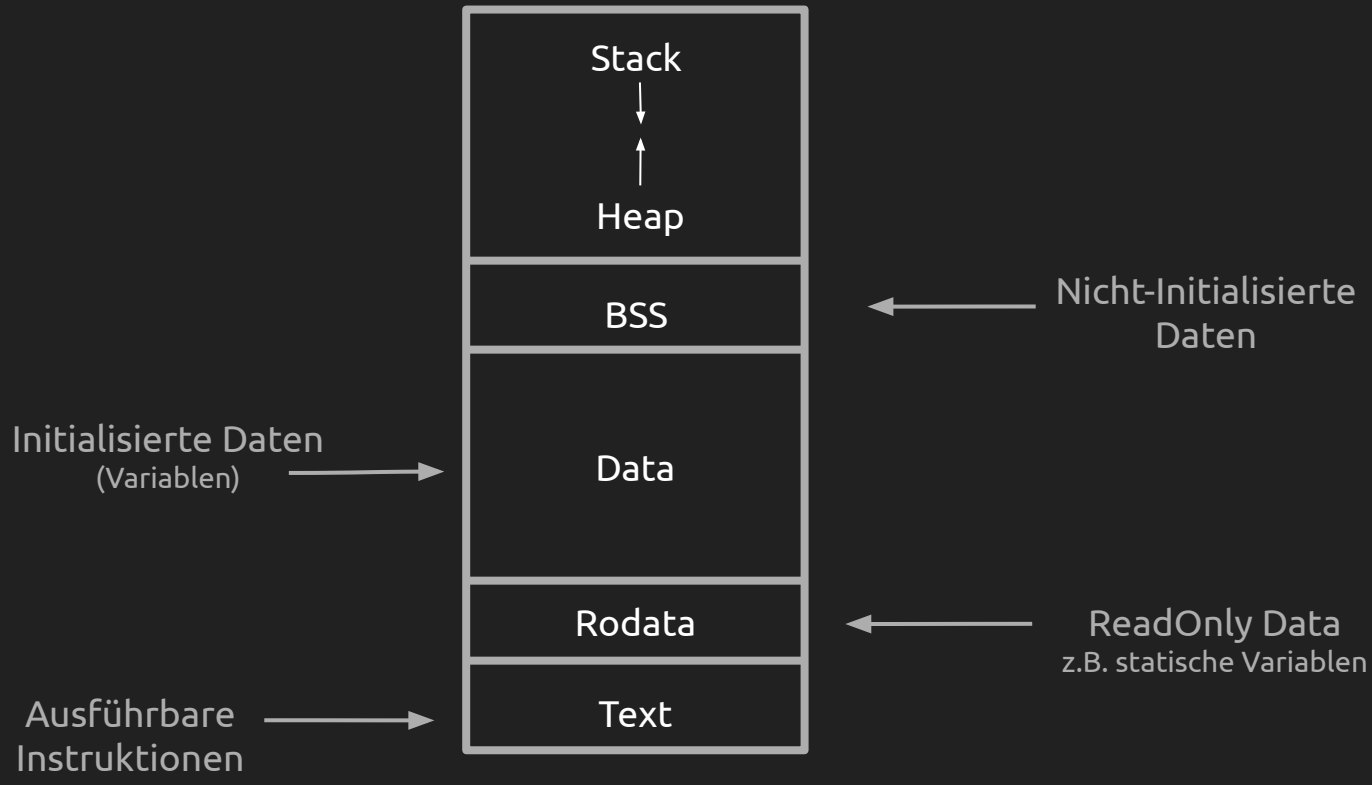
Demand Paging: How it works



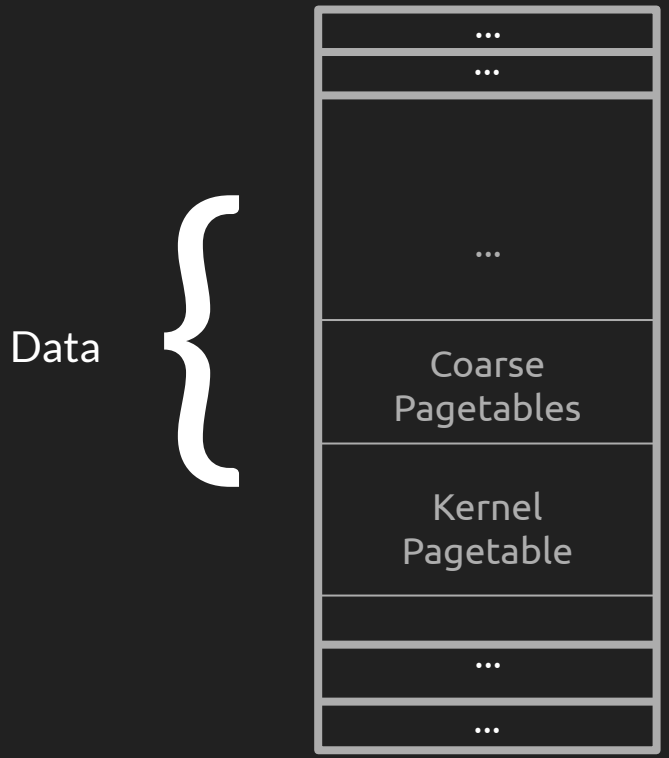
Wie viel Speicher wird beim initialisieren benötigt?

→ **Wir benötigen erstmal ein Memory Layout**

Memory Layout



Memory Layout



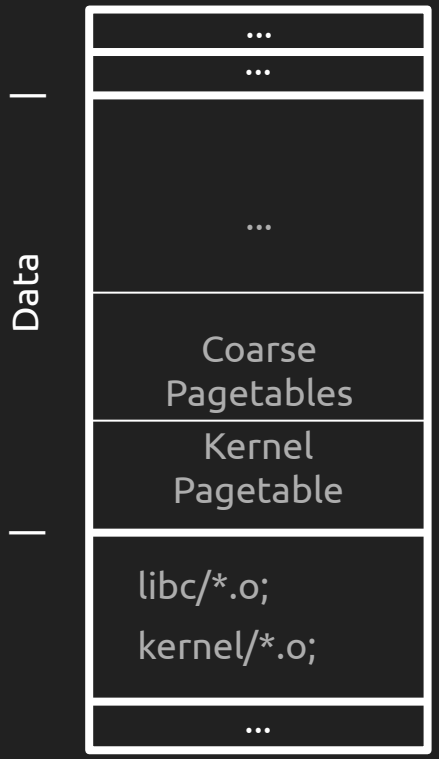
Initial benötigte Memory-Größe

- Wollen nur minimal viele Seiten initialisieren

  → Sind genau die Seiten, die für Memory Management benötigt werden

- Ansatz 1: Statisch Speicherbereiche festlegen
- Ansatz 2: Dynamisch

Memory Layout – Dynamisch




```
.minimal_memory_management_part  :
{
  . = ALIGN(4K);
  _minimal_memory_management_part_start  = .;

  libc/\*.o;
  kernel/\*.o;

  _minimal_memory_management_part_end  = .;
}

.data  :
{
  *(.data)

  . = ALIGN(16K);
  *(.kernel_page_table )

  . = ALIGN(1K);
  *(.all_coarse_page_tables )
}
```

Linker-Script

What’s next?

Ausblick

- Memory Layout umsetzen für weniger initial gemappten Speicher
- Effiziente Page-Verwaltung mit Listen (Free-Page-List etc.)
- Prozess-Isolation durch eigene Translation Tables
- Access Control für Pages (read/write/execute)
- Swapping (Pages bei Speicherknappheit auslagern)
- Shared Memory zwischen Prozessen

Quellen

- developer.arm.com/documentation/ddi0198/e
- imgflip.com/memegenerator
- fontawesome.com/icons
- asciiart.eu
- i.imgur.com/nLnX7BX.jpg

zul. abgerufen am 03.08.2020