Mini-Ulix: Übungsblatt 9

Hans-Georg Eßer, FOM Hochschule

26. Mai 2015

Inhaltsverzeichnis

1 Kernel-Layout **2**

2 Speicher einrichten: Die GDT **2**

3 Paging **4**

3.1 Page Directory und Page Table 4

3.2 Funktionen für den Zugriff 6

3.3 Initialisierung des Paging 7

4 Verwaltung der Rahmen **8**

4.1 Datenstruturen 8

4.2 Funktionen für den Zugriff 8

5 Verwaltung der Seiten **10**

5.1 Den physischen Speicher mappen 13

6 Hilfsfunktionen **14**

6.1 Speicher initialisieren mit `memset` 14

6.2 Text mit `printf` ausgeben 14

6.3 Den Bildschirm mit `clrscr` löschen 15

6.4 String kopieren mit `strncpy` 16

6.5 Hexdump 16

7 Interrupts **18**

8 Faults **24**

9 Tastatur-Treiber **27**

1 Kernel-Layout

Das Layout der C-Quellcode-Datei `ulix.c` unseres Mini-Kernels sieht schon – grob – wie beim richtigen ULIX aus:

2a ⟨*ulix.c* 2a⟩≡
```
⟨constants 7b⟩
⟨type definitions 2d⟩
⟨global variables 2c⟩
⟨macros 7a⟩
⟨function prototypes 3b⟩
⟨function implementations 3e⟩
⟨kernel main 2b⟩
```

Auf einen Abdruck der Assembler-Datei `start.asm` verzichten wir in dieser Ausarbeitung.

Die `main`-Funktion hat den folgenden Aufbau:

2b ⟨*kernel main* 2b⟩≡ (2a)
```
int main () {
  ⟨kernel main: initialize variables 14e⟩
  printf ("[1] entering main()\n");

  // Speicherverwaltung
  ⟨kernel main: setup paging 7c⟩
  ⟨kernel main: install GDT 4a⟩  // replace "trick GDT" with regular GDT
  paging_ready = true; printf ("[4] regular GDT is active\n");
  ⟨kernel main: map physical RAM 13c⟩
  ⟨kernel main: setup frame table 14a⟩

  ⟨kernel main: initialize system 21c⟩   // <-  !!
  ⟨kernel main: user-defined tests 27f⟩
  for (;;);   // inifinite loop
}
```

Die ersten Code-Chunks befassen sich mit dem Einrichten der Speicherverwaltung; dieses Thema haben wir in der Vorlesung nicht behandelt. Für uns interessant wird es erst im Code Chunk ⟨*kernel main: initialize system* 21c⟩. Springen Sie zu Kapitel 7 (ab Seite 18).

2c ⟨*global variables* 2c⟩≡ (2a) 3a▷
```
int paging_ready = false;
```

2 Speicher einrichten: Die GDT

Zunächst brauchen wir die Datenstrukturen. Die GDT-Einträge haben die in der Vorlesung vorgestellte Struktur, bei der die Werte `base` und `limit` umständlich auf mehrere Komponenten verteilt sind.

Für den Zugriff auf eine GDT braucht es zudem immer eine Art Zeiger auf den Tabellenanfang, der auch die Größe der GDT festlegt (Typ `struct gdt_ptr`).

2d ⟨*type definitions* 2d⟩≡ (2a) 4b▷

```
struct gdt_entry {
  unsigned int limit_low    : 16;
  unsigned int base_low     : 16;
  unsigned int base_middle  :  8;
  unsigned int access       :  8;
  unsigned int flags        :  4;
  unsigned int limit_high   :  4;
  unsigned int base_high    :  8;
};

struct gdt_ptr {
  unsigned int limit        : 16;
  unsigned int base         : 32;
} __attribute__((packed));
```

Hier kommt die GDT in den Speicher:

3a ⟨*global variables* 2c⟩+≡ (2a) ◁2c 5b▷

```
struct gdt_entry gdt[6];
struct gdt_ptr gp;
```

Die GDT zu "flushen" ist eine Aufgabe für eine Funktion, die in der Assembler-Datei definiert ist. Wir definieren Sie hier als extern:

3b ⟨*function prototypes* 3b⟩≡ (2a) 3c▷

```
extern void gdt_flush();
```

Um einen GDT-Eintrag mit Werten zu füllen, verwenden wir die Funktion

3c ⟨*function prototypes* 3b⟩+≡ (2a) ◁3b 3d▷

```
void gdt_set_gate (int num, unsigned long base,
  unsigned long limit, unsigned char access, unsigned char gran);
```

die `base` und `limit` aufteilt und zusammen mit den Zugriffsrechten (`access`) und den Flags (in `gran`) an den richtigen Stellen einträgt. Wir benutzen Sie bei der Initialisierung mit der Funktion

3d ⟨*function prototypes* 3b⟩+≡ (2a) ◁3c 6a▷

```
void gdt_install ();
```

welche die drei benötigten Segmentdeskriptoren erzeugt (einen Nulldeskriptor und die beiden echten Deskriptoren für das Code- und das Daten-Segment).

Hier sind die Implementierungen:

3e ⟨*function implementations* 3e⟩≡ (2a) 6c▷

```
void gdt_set_gate(int num, unsigned long base, unsigned long limit,
                  unsigned char access, unsigned char gran) {
  /* Setup the descriptor base address */
  gdt[num].base_low = (base & 0xFFFF);            // 16 bits
  gdt[num].base_middle = (base >> 16) & 0xFF;     //  8 bits
  gdt[num].base_high = (base >> 24) & 0xFF;       //  8 bits

  /* Setup the descriptor limits */
  gdt[num].limit_low  = (limit & 0xFFFF);         // 16 bits
  gdt[num].limit_high = ((limit >> 16) & 0x0F);   //  4 bits
```

```
  /* Finally, set up the granularity and access flags */
  gdt[num].flags = gran & 0xF;
  gdt[num].access = access;
}

void gdt_install() {
  gp.limit = (sizeof(struct gdt_entry) * 6) - 1;
  gp.base = (int) &gdt;

  gdt_set_gate(0, 0, 0, 0, 0);    // NULL descriptor

  // code segment
  gdt_set_gate(1, 0, 0xFFFFFFFF, 0b10011010, 0b1100 /* 0xCF */);

  // data segment
  gdt_set_gate(2, 0, 0xFFFFFFFF, 0b10010010, 0b1100 /* 0xCF */);

  gdt_flush();
}
```

Im Hauptprogramm (`main`) rufen wir `gdt_install()` dann später auf:

4a ⟨*kernel main: install GDT* 4a⟩≡ (2b)

```
gdt_install ();
```

3 Paging

Wir beginnen wieder mit den benötigten Datenstrukturen.

3.1 Page Directory und Page Table

Wir definieren zwei Datenstrukturen:

- Auf der obersten Ebene gibt es das *Page Directory* (`page_directory`), das aus 1024 *Page Table Descriptors* (`page_table_desc`) besteht:

4b ⟨*type definitions* 2d⟩+≡ (2a) ◁2d 5a▷

```
typedef struct {
  unsigned int present         : 1;  //  0
  unsigned int writeable       : 1;  //  1
  unsigned int user_accessible : 1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int undocumented    :  1;  //  6
  unsigned int zeroes          :  2;  //  8.. 7
  unsigned int unused_bits     :  3;  // 11.. 9
  unsigned int frame_addr      : 20;  // 31..12
} page_table_desc;

typedef struct { page_table_desc ptds[1024]; } page_directory;
```

Definiert:
`page_directory`, benutzt im Teil 5b.
`page_table_desc`, benutzt im Teil 6.

- Jeder Page Table Descriptor zeigt auf eine *Page Table* (`page_table`), welche wiederum aus 1024 *Page Descriptors* (`page_desc`) besteht:

5a ⟨*type definitions* 2d⟩+≡ (2a) ◁4b 10a▷

```
typedef struct {
  unsigned int present         : 1;  //  0
  unsigned int writeable       : 1;  //  1
  unsigned int user_accessible : 1;  //  2
  unsigned int pwt             :  1;  //  3
  unsigned int pcd             :  1;  //  4
  unsigned int accessed        :  1;  //  5
  unsigned int dirty           :  1;  //  6
  unsigned int zeroes          :  2;  //  8.. 7
  unsigned int unused_bits     :  3;  // 11.. 9
  unsigned int frame_addr      : 20;  // 31..12
} page_desc;

typedef struct { page_desc pds[1024]; } page_table;
```

Definiert:
`page_desc`, benutzt im Teil 6.
`page_table`, benutzt im Teils 5b, 10c, 12, und 13b.

- Die Page Descriptors zeigen schließlich auf die *Page Frames*, also 4 KByte große Blöcke im Hauptspeicher, welche unsere Seiten aufnehmen.

In dieser Miniversion von ULIX verwenden wir nur jeweils ein einziges Page Directory und 17 Page Tables, für die wir zunächst Speicher reservieren und dann noch Pointer darauf einrichten:

5b ⟨*global variables* 2c⟩+≡ (2a) ◁3a 8b▷

```
page_directory kernel_pd     __attribute__ ((aligned (4096)));
page_table kernel_pt         __attribute__ ((aligned (4096)));
page_table kernel_pt_ram[16] __attribute__ ((aligned (4096)));

page_directory* current_pd = &kernel_pd;
page_table*     current_pt = &kernel_pt;
```

Definiert:
`current_pd`, benutzt im Teils 7, 8a, 10c, 12, und 13.
`current_pt`, benutzt im Teil 7.
`kernel_pd`, nicht benutzt.
`kernel_pt`, nicht benutzt.
Benutzt `page_directory` 4b und `page_table` 5a.

Hier ist wichtig, dass die Datenstrukturen über das Attribut `((aligned (4096)))` jeweils ab einem Vielfachen von 4096 (der Seitengröße) beginnen; sie dürfen nicht "quer" im Speicher liegen.

Das Page Directory `kernel_pd` und die Page Table `kernel_pt` werden wir gleich nutzen, um das Paging an sich zu aktivieren; die 16 Page Tables im Array `kernel_pt_ram` brauchen wir später, um den Hauptspeicher (64 MByte) nach `0xD0000000`... zu mappen.

3.2 Funktionen für den Zugriff

Wir definieren zwei Funktionen

6a ⟨*function prototypes* 3b⟩+≡ (2a) ◁3d 6b▷
```
page_table_desc* fill_page_table_desc (page_table_desc *ptd,
  unsigned int present, unsigned int writeable,
  unsigned int user_accessible, unsigned int frame_addr);
```
Benutzt page_table_desc 4b.

und

6b ⟨*function prototypes* 3b⟩+≡ (2a) ◁6a 9a▷
```
page_desc* fill_page_desc (page_desc *pd, unsigned int present,
  unsigned int writeable, unsigned int user_accessible,
  unsigned int dirty, unsigned int frame_addr);
```
Benutzt page_desc 5a.

mit denen wir einen Page Table Descriptor (oberste Ebene) bzw. einen Page Descriptor (zweite Ebene) mit Inhalten füllen können. Die beiden Funktionen sind ähnlich aufgebaut und unterscheiden sich nur darin, dass `fill_page_desc()` ein zusätzliches Argument akzeptiert, mit dem das `dirty`-Flag gesetzt werden kann: Das gibt es nur bei Page Descriptors, der Page Table Descriptor hat an derselben Stelle einen undokumentierten Eintrag, den wir immer auf 0 setzen.

Hier sind die Implementierungen:

6c ⟨*function implementations* 3e⟩+≡ (2a) ◁3e 9c▷
```
page_desc* fill_page_desc (page_desc *pd, unsigned int present,
  unsigned int writeable, unsigned int user_accessible,
  unsigned int dirty, unsigned int frame_addr) {

  memset (pd, 0, sizeof(pd));

  pd->present = present;
  pd->writeable = writeable;
  pd->user_accessible = user_accessible;
  pd->dirty = dirty;
  pd->frame_addr = frame_addr >> 12;   // right shift, 12 bits
  return pd;
};

page_table_desc* fill_page_table_desc (page_table_desc *ptd,
  unsigned int present, unsigned int writeable,
  unsigned int user_accessible, unsigned int frame_addr) {

  memset (ptd, 0, sizeof(ptd));

  ptd->present = present;
  ptd->writeable = writeable;
  ptd->user_accessible = user_accessible;
  ptd->frame_addr = frame_addr >> 12;   // right shift, 12 bits
  return ptd;
};
```
Benutzt page_desc 5a und page_table_desc 4b.

Die hier verwendete Funktion `memset` beschreiben wir am Ende dieses Textes.

Um beide Funktionen leichter mit Standardparametern aufrufen zu können, führen wir noch zwei Makros

7a ⟨*macros* 7a⟩≡ (2a) 8d▷
```
#define KMAP(pd,frame) \
  fill_page_desc (pd, true, true, false, false, frame)
#define KMAPD(ptd, frame) \
  fill_page_table_desc (ptd, true, true, false, frame)
```

ein, die Einträge für die Nutzung durch den Kernel erzeugen. Später, wenn wir den User Mode einführen, wird es ganz ähnlich aussehende Makros `UMAP` und `UMAPD` geben, die sich nur darin von `KMAP` und `KMAPD` unterscheiden, dass sie `user_accessible` jeweils auf `true` statt `false` setzen. Übrigens müssen wir diese beiden booleschen Konstanten auch erst definieren:

7b ⟨*constants* 7b⟩≡ (2a) 8c▷
```
#define false 0
#define true 1
```

3.3 Initialisierung des Paging

Beim Paging richten wir die Seitentabellen zunächst so ein, dass sie ordentlich mit unserer Trick-GDT zusammen spielen, welche über einen Base-Wert von `0x40000000` alle Adressen ab `0xC0000000` in Adressen ab `0` umrechnet.

Zunächst bereiten wir das Page Directory `current_pd` vor, indem wir es mit Null-Einträgen füllen:

7c ⟨*kernel main: setup paging* 7c⟩≡ (2b) 7d▷
```
for (int i=1; i<1024; i++) {
  fill_page_table_desc (&(current_pd->ptds[i]), false, false, false, 0);
};
```
Benutzt `current_pd` 5b.

Dabei lassen wir den ersten Eintrag aus, den wir separat bearbeiten: Wir lassen den Eintrag `0` und den Eintrag `768` auf die Seitentabelle `current_pt` zeigen, damit wir zwei Mappings auf die ersten 4 MB RAM haben, einmal ab `0` und einmal ab `0xC0000000`:

7d ⟨*kernel main: setup paging* 7c⟩+≡ (2b) ◁7c 7e▷
```
KMAPD ( &(current_pd->ptds[  0]), (unsigned int)(current_pt)-0xC0000000 );
KMAPD ( &(current_pd->ptds[768]), (unsigned int)(current_pt)-0xC0000000 );
```
Benutzt `current_pd` 5b und `current_pt` 5b.

In der Seitentabelle `current_pt` tragen wir für die ersten 1023 Rahmen das Mapping ein:

7e ⟨*kernel main: setup paging* 7c⟩+≡ (2b) ◁7d 8a▷
```
for (int i=0; i<1023; i++) {
  KMAP ( &(current_pt->pds[i]), i*4096 );
};
printf ("[2] page directory setup, with identity mapping\n");
```
Benutzt `current_pt` 5b.

Schließlich aktivieren wir das Paging, indem wir die Control Registers `cr0` und `cr3` mit passenden Werten füllen bzw. aktualisieren:

8a ⟨*kernel main: setup paging* 7c⟩+≡ (2b) ◁7e

```
unsigned int cr0;
char *kernel_pd_address;
kernel_pd_address = (char*)(current_pd) - 0xC0000000;
asm volatile ("mov %0, %%cr3" : : "r"(kernel_pd_address));
  // write CR3
asm volatile ("mov %%cr0, %0" : "=r"(cr0) : );  // read  CR0
cr0 |= (1<<31);      // Enable paging by setting PG bit 31 of CR0
asm volatile ("mov %0, %%cr0" : : "r"(cr0) );   // write CR0
printf ("[3] paging activated.\n");
```

Benutzt `current_pd` 5b.

4 Verwaltung der Rahmen

Bisher haben wir nur gezeigt, wie das System das Paging vorbereitet und aktiviert – im regulären Betrieb müssen wir aber auch dynamisch neuen Speicher allozieren. Es wird also nötig sein, Seiten anzufordern und wieder freizugeben. Als ersten Schritt in diese Richtung brauchen wir eine Verwaltung des physischen Speichers: Hier geht es um freie Seitenrahmen (*Frames*).

4.1 Datenstruturen

Wir legen eine Rahmentabelle `ftable` an, die für jeden der 64 MByte / 4 KByte = 16 K Rahmen des Hauptspeichers ein Bit enthält: Ist es `0`, ist der Rahmen frei; anderenfalls ist er belegt. Die Anzahl der noch verfügbaren Rahmen merken wir uns in einer Variablen `free_frames`:

8b ⟨*global variables* 2c⟩+≡ (2a) ◁5b 16a▷

```
unsigned int free_frames = NUMBER_OF_FRAMES;
char place_for_ftable[NUMBER_OF_FRAMES/8];
unsigned int* ftable = (unsigned int*)(&place_for_ftable);
```

Hier benutzen wir gleich zweimal die Konstante `NUMBER_OF_FRAMES`, die wir bisher nicht erwähnt haben. Sie berechnet sich, wie oben beschrieben, aus der Speichergröße `MEM_SIZE` und der Größe einer Seite `PAGE_SIZE`:

8c ⟨*constants* 7b⟩+≡ (2a) ◁7b 17b▷

```
#define MEM_SIZE 1024*1024*64        // 64 MByte
#define MAX_ADDRESS MEM_SIZE-1       // last valid physical address
#define PAGE_SIZE 4096               // Intel: 4K pages
#define NUMBER_OF_FRAMES MEM_SIZE/PAGE_SIZE
```

4.2 Funktionen für den Zugriff

Um auf die einzelnen Bits in `ftable` zugreifen zu können, definieren wir zunächs zwei Helfer-Makros:

8d ⟨*macros* 7a⟩+≡ (2a) ◁7a 11a▷

```
#define INDEX_FROM_BIT(b) (b/32)   // 32 bits in an unsigned int
#define OFFSET_FROM_BIT(b) (b%32)
```

`INDEX_FROM_BIT` findet für eine Rahmennummer zunächst heraus, in welchem der je 32 Bit breiten Integers der Tabelle sich das Bit befindet; im zweiten Schritt berechnet `OFFSET_FROM_BIT` dann die Position innerhalb dieses Integers. Mit diesen beiden Helfer-Makros können wir nun drei Funktionen

9a ⟨*function prototypes* 3b⟩+≡ (2a) ◁6b 9d▷

```
static void set_frame (unsigned int frame_addr);
static void clear_frame (unsigned int frame_addr);
static unsigned int test_frame (unsigned int frame_addr);
```

implementieren, mit denen wir einzelne Bits setzen, zurücksetzen oder abfragen können. Alle nutzen zunächst die Makros, um aus der Frame-Adresse `frame_addr` die zwei Werte `index` und `offset` zu berechnen

9b ⟨*calculate index and offset* 9b⟩≡ (9c)

```
unsigned int frame = frame_addr / PAGE_SIZE;
unsigned int index  = INDEX_FROM_BIT  (frame);
unsigned int offset = OFFSET_FROM_BIT (frame);
```

und dann das jeweilige Bit zu verändern oder auszulesen:

9c ⟨*function implementations* 3e⟩+≡ (2a) ◁6c 9e▷

```
static void set_frame (unsigned int frame_addr) {
  ⟨calculate index and offset 9b⟩
  ftable[index] |= (1 << offset);
}

static void clear_frame (unsigned int frame_addr) {
  ⟨calculate index and offset 9b⟩
  ftable[index] &= ~(1 << offset);
}

static unsigned int test_frame (unsigned int frame_addr) {
  // returns true if frame is in use (false if frame is free)
  ⟨calculate index and offset 9b⟩
  return ((ftable[index] & (1 << offset)) >> offset);
}
```

Damit ist der Zugriff auf die Frame-Tabelle geregelt, jetzt können wir Funktionen entwickeln, mit denen der Kernel explizit einen neuen Frame (für die eigene Nutzung) anfordert oder wieder freigibt. Wir nennen diese Funktionen

9d ⟨*function prototypes* 3b⟩+≡ (2a) ◁9a 10b▷

```
int request_new_frame ();
void release_frame (unsigned int frameaddr);
```

– die letzte der beiden nimmt eine physische Speicheradresse als Argument, wie sie von der ersten zurück gegeben wird. Der Aufruf

```
release_frame ( request_new_frame () );
```

sollte also neutral sein. Zur Implementierung ist nur bei `request_new_frame()` ein wenig Arbeit nötig, weil hier die Frame-Tabelle durchsucht wird:

9e ⟨*function implementations* 3e⟩+≡ (2a) ◁9c 10c▷

```
int request_new_frame () {
  unsigned int frameid;
  boolean found=false;
  for (frameid = 0; frameid < NUMBER_OF_FRAMES; frameid++) {
    if ( !test_frame (frameid*4096) ) {
      found=true;
      break;   // frame found
    };
  }
  if (found) {
    memset ((void*)PHYSICAL(frameid << 12), 0, PAGE_SIZE);
    set_frame (frameid*4096);
    free_frames--;
    return frameid;
  } else {
    return -1;
  }
};

void release_frame (unsigned int frameaddr) {
  if ( test_frame (frameaddr) ) {
    // only do work if frame is marked as used
    clear_frame (frameaddr);
    free_frames++;
  };
};
```

Den hier benutzten Typ `boolean` müssen wir noch deklarieren:

10a ⟨*type definitions* 2d⟩+≡ (2a) ◁5a 19d▷

```
typedef unsigned int boolean;
```

5 Verwaltung der Seiten

Ohne Rahmen keine Seiten: Nachdem nun die Verwaltung der Frames funktioniert, können wir uns der Vergabe von Seiten zuwenden.

Die Datenstrukturen für das Paging sind schon vorhanden, die haben wir bei der Initialisierung des Pagings besprochen. Hier geht es nun darum, im laufenden Betrieb dynamisch neue Seiten anzufordern und diese wieder zurückzugeben – beides ist nur möglich, indem auch Frames angefordert und freigegeben werden, und wir müssen dazu auch das Page Directory und die Page Tables überarbeiten sowie ggf. neue Page Tables erzeugen.

Wir starten mit der einfachen Funktion

10b ⟨*function prototypes* 3b⟩+≡ (2a) ◁9d 11b▷

```
unsigned int pageno_to_frameno (unsigned int pageno);
```

welche für bereits "gemappten" virtuellen Speicher eine Seitennummer in die zugehörige Rahmennummer umrechnet. Das funktioniert genauso wie in der MMU (Memory Management Unit):

10c ⟨*function implementations* 3e⟩+≡ (2a) ◁9e 11c▷

```
unsigned int pageno_to_frameno (unsigned int pageno) {
  unsigned int pdindex = pageno/1024;
  unsigned int ptindex = pageno%1024;
  if ( ! current_pd->ptds[pdindex].present ) {
    return -1;       // we don't have that page table
  } else {
    // get the page table
    page_table* pt = (page_table*)
      ( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
    if ( pt->pds[ptindex].present ) {
      return pt->pds[ptindex].frame_addr;
    } else {
      return -1;     // we don't have that page
    };
  };
};
```

Benutzt `current_pd` 5b und `page_table` 5a.

Die Funktion verwendet das Makro `PHYSICAL`, das einfach zu jeder Adresse den Wert `0xD0000000` addiert, um über das Mapping des physischen Speichers in den virtuellen Adressraum (ab `0xD0000000`) auf den Hauptspeicher zuzugreifen:

11a ⟨*macros* 7a⟩+≡ (2a) ◁8d 17a▷

```
#define PHYSICAL(x) ((x)+0xd0000000)
```

Wenn kein zugeordneter Frame gefunden wird (entweder weil es schon im Page Directory oder in der richtigen Page Table keinen Eintrag gibt), gibt die Funktion `-1` zurück.

Jetzt wird es kompliziert: Wir implementieren nun die Funktion

11b ⟨*function prototypes* 3b⟩+≡ (2a) ◁10b 13a▷

```
unsigned int* request_new_page (int need_more_pages);
```

die eine neue Seite anfordert. Der Parameter `need_more_pages` wird in der aktuellen Version des Codes noch nicht ausgewertet; er dient später dazu, mehrere zusammenhängende Seiten anzufordern.

Die Funktion besorgt sich zunächst einen frischen Frame und sucht dann nach einer freien Seitennummer:

11c ⟨*function implementations* 3e⟩+≡ (2a) ◁10c 13b▷

```
unsigned int* request_new_page (int need_more_pages) {
  ⟨page request implementation 11d⟩
}
```

11d ⟨*page request implementation* 11d⟩≡ (11c) 12a▷

```
unsigned int newframeid = request_new_frame ();
if (newframeid == -1) { return NULL; }  // exit if no frame was found
unsigned int pageno = -1;
for (unsigned int i=0xc0000; i<1024*1024; i++) {
  if ( pageno_to_frameno (i) == -1 ) {
    pageno = i;
    break;       // end loop, unmapped page was found
  };
};
```

```
if ( pageno == -1 ) {
  return NULL;   // we found no page -- whole 4 GB are mapped???
};
```

An dieser Stelle haben wir den Frame und die Seitennummer. Jetzt müssen wir das neue Mapping eintragen. Dazu berechnen wir zunächst die Positionen im Page Directory und der jeweiligen Page Table:

12a ⟨*page request implementation* 11d⟩+≡ (11c) ◁11d 12b▷

```
unsigned int pdindex = pageno/1024;
unsigned int ptindex = pageno%1024;
page_table* pt;
```

Benutzt page_table 5a.

Wenn `ptindex == 0` gilt, müssen wir eine neue Seitentabelle "anbrechen". Wir gehen hier davon aus, dass wir diese zunächst erstellen müssen. (Tatsächlich müssten wir prüfen, ob sie schon existiert – das könnte passieren, wenn wir Speicher wieder freigeben und dann erneut anfordern.) Für die neue Seitentabelle verwenden wir dann den bereits angeforderten Frame und benötigen danach einen neuen.

12b ⟨*page request implementation* 11d⟩+≡ (11c) ◁12a 12c▷

```
if (ptindex == 0) {
  // last entry! // create a new page table in the reserved frame
  page_table* pt = (page_table*) PHYSICAL(newframeid<<12);
  memset (pt, 0, PAGE_SIZE);
  KMAPD ( &(current_pd->ptds[pdindex]), newframeid << 12 );

  newframeid = request_new_frame ();  // get yet another frame
  if (newframeid == -1) {
    return NULL;                      // exit if no frame was found
    // note: we're not removing the new page table since we assume
    // it will be used soon anyway
  }
};
```

Benutzt current_pd 5b und page_table 5a.

Weiter geht es mit dem Eintragen des Mappings, dazu suchen wir zunächst die richtige Seitentabelle heraus (die über den Index `pdindex` im Page Directory festgelegt ist) und tragen an deren Position `ptindex` den Frame ein, wobei uns das Makro `KMAP` hilft:

12c ⟨*page request implementation* 11d⟩+≡ (11c) ◁12b

```
pt = (page_table*)( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
// finally: enter the frame address
KMAP ( &(pt->pds[ptindex]), newframeid * PAGE_SIZE );

// invalidate cache entry
asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );

memset ((unsigned int*) (pageno*4096), 0, 4096);
return ((unsigned int*) (pageno*4096));
```

Benutzt current_pd 5b und page_table 5a.

(Am Ende invalidieren wir mit der CPU-Instruktion `invlpg` einen eventuell vorhandenen Eintrag im Cache der MMU, initialisieren die neue Seite, so dass sie nur Nullen enthält, und geben ihre (virtuelle) Adresse zurück.

Eine Seite wieder freizugeben, ist leichter: Die Funktion

13a ⟨*function prototypes* 3b⟩+≡ (2a) ◁11b 14b▷
```
void release_page (unsigned int pageno);
```

die eine Seitennummer als Argument erwartet, ist schnell implementiert: Sie ersetzt den von `request_new_frame` eingetragenen Page Descriptor wieder durch einen Null-Deskriptor und gibt auch den zugeordneten Frame frei.

13b ⟨*function implementations* 3e⟩+≡ (2a) ◁11c 14c▷
```
void release_page (unsigned int pageno) {
  int frameno = pageno_to_frameno (pageno);  // we will need this later
  if ( frameno == -1 )  { return; }          // exit if no such page
  unsigned int pdindex = pageno/1024;
  unsigned int ptindex = pageno%1024;
  page_table* pt;
  pt = (page_table*)
    ( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
  // write null page descriptor
  memset (&(pt->pds[ptindex]), 0, 4);
  fill_page_desc (&(pt->pds[ptindex]), false, false, false, false, 0);
  release_frame (frameno<<12);   // expects an address, not an ID
  asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
  // gdt_flush ();
};
```
Benutzt `current_pd` 5b und `page_table` 5a.

Auch hier wird am Ende wieder `invlpg` verwendet, um eventuelle Informationen über diese Seite im MMU-Cache zu löschen.

5.1 Den physischen Speicher mappen

Das Mapping des physischen Speichers in den Adressbereich ab `0xD0000000` funkioniert so, dass wir jeweils für die virtuelle Seite `x + 0xD0000` den physischen Frame `x` eintragen. Dafür benötigen wir die bereits oben erwähnten 16 Seitentabellen:

13c ⟨*kernel main: map physical RAM* 13c⟩≡ (2b)
```
memset (kernel_pt_ram, 0, 4);

for (unsigned int fid=0; fid<NUMBER_OF_FRAMES; fid++) {
  KMAP ( &(kernel_pt_ram[fid/1024].pds[fid%1024]), fid*PAGE_SIZE );
}
unsigned int physaddr;
for (int i=0; i<16; i++) {
  // get physical address of kernel_pt_ram[i]
  physaddr = (unsigned int)(&(kernel_pt_ram[i])) - 0xc0000000;
  KMAPD ( &(current_pd->ptds[832+i]), physaddr );
};

gdt_flush ();
```
Benutzt `current_pd` 5b.

Das Einrichten der Frame-Tabelle besteht nur darin, passende `0`- und `1`-Bits hineinzuschreiben. Dafür können wir zweimal `memset` verwenden: Der erste Aufruf füllt die Tabelle mit Nullen, und der zweite setzt die vordersten $128 \times 8 = 1024$ Bits auf `1`, weil die ersten 1024 Rahmen (also die ersten 4 MByte) nicht von der Speicherverwaltung verwendet werden sollen.

14a ⟨*kernel main: setup frame table* 14a⟩≡ (2b)

```
memset (ftable, 0, NUMBER_OF_FRAMES/8);  // all frames are free
memset (ftable, 0xff, 128);
free_frames -= 1024;
```

6 Hilfsfunktionen

Hier finden Sie alle Funktionen, die eher uninteressant sind – z. B., weil sie klassische Hilfsfunktionen sind, die sonst von Bibliotheken bereitgestellt werden.

6.1 Speicher initialisieren mit `memset`

Die Funktionen

14b ⟨*function prototypes* 3b⟩+≡ (2a) ◁13a 14d▷

```
void *memset  (void *dest, char val,  int count);
void *memsetw (void *dest, short val, int count);
```

erwarten als Argumente eine Startadresse, ein Füll-Byte oder Füll-Wort und die Anzahl der zu füllenden Bytes. Sie ist schnell geschrieben:

14c ⟨*function implementations* 3e⟩+≡ (2a) ◁13b 15b▷

```
void *memset (void *dest, char val, int count) {
  char *temp = (char *)dest;
  for( ; count != 0; count--) *temp++ = val;
  return dest;
}

void *memsetw (void *dest, short val, int count) {
  short *temp = (short *)dest;
  for( ; count != 0; count--) *temp++ = val;
  return dest;
}
```

6.2 Text mit `printf` ausgeben

Die Funktion

14d ⟨*function prototypes* 3b⟩+≡ (2a) ◁14b 15a▷

```
extern int printf(const char *format, ...);
```

stellen wir über eine separate C-Datei bereit, deren Inhalt wir hier nicht weiter erklären. Damit sie funktioniert, müssen wir zwei Variablen initialisieren:

14e ⟨*kernel main: initialize variables* 14e⟩≡ (2b)

```
posx = 0; posy = 8;  // set cursor
```

Die `printf`-Funktion nutzt die einfachere Funktion

15a ⟨*function prototypes* 3b⟩+≡ (2a) ◁14d 15c▷
```
void kputch (char c);
```

welche ein einzelnes Zeichen auf dem Bildschirm ausgeben kann. Sie schreibt direkt in den Textmodus-Framebuffer der Grafikkarte, welcher in den physischen Adressraum eingeblendet ist. Um sich die aktuelle Cursorposititon zu merken, verwendet sie die Variablen `posx` und `posy`.

15b ⟨*function implementations* 3e⟩+≡ (2a) ◁14c 15e▷
```
void kputch (char c) {
  char *screen;

  if (c=='\n') {
    posy ++;
    posx = 0;
    uartputc ('\n');
    return;
  }

  if (paging_ready)
    screen = (char*) 0xb8000 + posy*160 + posx*2;
  else
    screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
  *screen = c;
  posx++;
  if (posx == 80) {
    posy++; posx = 0;
  }

  // auf serielle Konsole schreiben; ohne Erklärung
  if (c == 0x100) {  //  backspace
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else uartputc(c);
}
```

Die Funktion erzeugt oben für Zeilenumbrüche und auch am Ende auch eine Ausgabe über die serielle Konsole, was wir hier nicht weiter erklären; die Funktion `uartputc` müssen wir allerdings als extern deklarieren:

15c ⟨*function prototypes* 3b⟩+≡ (2a) ◁15a 15d▷
```
extern void uartputc (int c);
```

6.3 Den Bildschirm mit `clrscr` löschen

Wenn Sie mehr Platz auf dem Bildschirm (für weitere Ausgaben) brauchen, können Sie durch einen Aufruf der Funktion

15d ⟨*function prototypes* 3b⟩+≡ (2a) ◁15c 16b▷
```
void clrscr ();
```

den Bildschirm löschen; das setzt auch den Cursor automatisch nach links oben:

15e ⟨*function implementations* 3e⟩+≡ (2a) ◁15b 16c▷

```
void clrscr () {
  posx = posy = 0;
  unsigned blank = 0x20 + (0x0f<<8);   // blank character (word)
  char *screen;
  if (paging_ready)
    screen = (char*) 0xb8000;
  else
    screen = (char*) 0xc0000000 + 0xb8000;
  memsetw (screen, blank, 80*25);
}
```

Die beiden hier verwendeten Variablen für die Cursor-Position müssen wir noch deklarieren:

16a ⟨*global variables* 2c⟩+≡ (2a) ◁8b 20a▷
```
int posx, posy;
```

6.4 String kopieren mit `strncpy`

16b ⟨*function prototypes* 3b⟩+≡ (2a) ◁15d 16d▷
```
void *strncpy(void *dest, const void *src, int count);
```

16c ⟨*function implementations* 3e⟩+≡ (2a) ◁15e 16e▷
```
void *strncpy (void *dest, const void *src, int count) {
  // like memcpy, but copies only until first \0 character
  const char *sp = (const char *)src;
  char *dp = (char *)dest;
  for (; count != 0; count--) {
    *dp = *sp;
    if (*dp == 0) break;
    dp++; sp++;
  }
  return dest;
}
```

6.5 Hexdump

Zum Bearbeiten der Übungsaufgabe stellen wir noch die folgende Funktion

16d ⟨*function prototypes* 3b⟩+≡ (2a) ◁16b 18b▷
```
void hexdump (unsigned int start, unsigned int end);
```

bereit, welche einen Hexdump des virtuellen Speichers zwischen `start` und `end` erzeugt:

16e ⟨*function implementations* 3e⟩+≡ (2a) ◁16c 18c▷
```
void hexdump (unsigned int start, unsigned int end) {
  char z;
  for (unsigned int i=start; i < end; i+=16) {
    printf ("%x  ", i);  // address
    // hex values
    for (int j=i; j<i+16; j++) {
      printf ("%02x ", (unsigned char)PEEK(j));
```

```
      if (j==i+7) kputch (' ');
    };
    kputch (' ');
    // char values
    for (int j=i; j<i+16; j++) {
      z = PEEK(j);
      if ((z>32)&&(z<127)) {
        kputch (PEEK(j));
      } else {
        kputch ('.');
      }
    }

    kputch ('\n');
  }
}
```

Sie verwendet das Makro `PEEK` zum Auslesen des Speichers:

17a ⟨*macros* 7a⟩+≡ (2a) ◁11a 24e▷

```
#define PEEK(addr) (*(unsigned char *)(addr))
```

Zum Abschluss noch drei Makros, die wir im Code verwendet haben:

17b ⟨*constants* 7b⟩+≡ (2a) ◁8c 18a▷

```
#define asm __asm__
#define volatile __volatile__
#define NULL ((void*) 0)
```

7 Interrupts

Wir implementieren nun die Interrupt- (und im nächsten Kapitel die Fault-) Handler und starten dabei mit den Interrupt-Nummern:

18a ⟨*constants* 7b⟩+≡ (2a) ◁17b 18d▷

```
#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_SLAVE      2     // Here the slave PIC connects to master
#define IRQ_COM2       3
#define IRQ_COM1       4
#define IRQ_FDC        6
#define IRQ_IDE       14     // primary IDE controller; secondary has IRQ 15
```

Definiert:
IRQ_COM1, nicht benutzt.
IRQ_COM2, nicht benutzt.
IRQ_FDC, nicht benutzt.
IRQ_IDE, nicht benutzt.
IRQ_KBD, nicht benutzt.
IRQ_SLAVE, benutzt im Teil 20e.
IRQ_TIMER, nicht benutzt.

Weiter geht es mit den `in`- und `out`-Befehlen

18b ⟨*function prototypes* 3b⟩+≡ (2a) ◁16d 20b▷

```
unsigned char  inportb (unsigned short port);
unsigned short inportw (unsigned short port);
void outportb (unsigned short port, unsigned char data);
void outportw (unsigned short port, unsigned short data);
```

Benutzt inportb 18c, inportw 18c, outportb 18c, und outportw 18c.

die wir wir in der Vorlesung implementieren; der Code für `inportb` und `outportb` steht bereits in der Datei `printf.c`, weswegen wir ihn hier weglassen.

18c ⟨*function implementations* 3e⟩+≡ (2a) ◁16e 20c▷

```
unsigned short inportw (unsigned short port) {
  unsigned short retval;
  asm volatile ("inw %%dx, %%ax" : "=a" (retval) : "d" (port));
  return retval;
}

void outportw (unsigned short port, unsigned short data) {
  asm volatile ("outw %%ax, %%dx" : : "d" (port), "a" (data));
}
```

Definiert:
inportb, benutzt im Teils 18b und 21b.
inportw, benutzt im Teil 18b.
outportb, benutzt im Teils 18, 19, 21b, und 23a.
outportw, benutzt im Teil 18b.

Damit können wir nun die PICs einrichten. Ihre Ports haben die folgenden Adressen:

18d ⟨*constants* 7b⟩+≡ (2a) ◁18a 23b▷

```
// I/O Addresses of the two programmable interrupt controllers
#define IO_PIC_MASTER_CMD   0x20  // Master (IRQs 0-7), command register
#define IO_PIC_MASTER_DATA  0x21  // Master, control register
```

```
#define IO_PIC_SLAVE_CMD    0xA0  // Slave (IRQs 8-15), command register
#define IO_PIC_SLAVE_DATA   0xA1  // Slave, control register
```

Definiert:
IO_PIC_MASTER_CMD, benutzt im Teils 19b und 23a.
IO_PIC_MASTER_DATA, benutzt im Teils 19 und 21b.
IO_PIC_SLAVE_CMD, benutzt im Teils 19b und 23a.
IO_PIC_SLAVE_DATA, benutzt im Teils 19 und 21b.

Wir müssen bei der Initialisierung die Interrupt-Nummern umbiegen; detaillierte Erläuterungen dazu finden Sie im Buch-Kapitel 12 auf der Webseite.

19a ⟨*remap the interrupts to 32..47* 19a⟩≡ (20e)
```
⟨PIC: program/initialize the PICs 19b⟩
⟨PIC: set the initial interrupt mask 19c⟩
```

19b ⟨*PIC: program/initialize the PICs* 19b⟩≡ (19a)
```
outportb (IO_PIC_MASTER_CMD,  0x11);  // ICW1: initialize; begin programming
outportb (IO_PIC_SLAVE_CMD,   0x11);  // ICW1: dito, for PIC2
outportb (IO_PIC_MASTER_DATA, 0x20);  // ICW2 for PIC1: offset 0x20
                                      // (remaps 0x00..0x07 -> 0x20..0x27)
outportb (IO_PIC_SLAVE_DATA,  0x28);  // ICW2 for PIC2: offset 0x28
                                      // (remaps 0x08..0x0f -> 0x28..0x2f)
outportb (IO_PIC_MASTER_DATA, 0x04);  // ICW3 for PIC1: there's a slave on IRQ 2
                                      // (0b00000100 = 0x04)
outportb (IO_PIC_SLAVE_DATA,  0x02);  // ICW3 for PIC2: your slave ID is 2
outportb (IO_PIC_MASTER_DATA, 0x01);  // ICW4 for PIC1 and PIC2: 8086 mode
outportb (IO_PIC_SLAVE_DATA,  0x01);
```

Benutzt IO_PIC_MASTER_CMD 18d, IO_PIC_MASTER_DATA 18d, IO_PIC_SLAVE_CMD 18d, IO_PIC_SLAVE_DATA 18d, und outportb 18c.

Im zweiten Schritt setzen wir die Interrupt-Maske (und schalten alle Interrupts aus).

19c ⟨*PIC: set the initial interrupt mask* 19c⟩≡ (19a)
```
outportb (IO_PIC_MASTER_DATA, 0x00);  // PIC1: mask 0
outportb (IO_PIC_SLAVE_DATA,  0x00);  // PIC2: mask 0
```

Benutzt IO_PIC_MASTER_DATA 18d, IO_PIC_SLAVE_DATA 18d, und outportb 18c.

Weiter geht es mit der Interrupt Descriptor Table. Ihre Einträge haben folgenden Aufbau:

19d ⟨*type definitions* 2d⟩+≡ (2a) ◁10a 19e▷
```
struct idt_entry {
    unsigned int addr_low  : 16;   // lower 16 bits of address
    unsigned int gdtsel    : 16;   // use which GDT entry?
    unsigned int zeroes    :  8;   // must be set to 0
    unsigned int type      :  4;   // type of descriptor
    unsigned int flags     :  4;
    unsigned int addr_high : 16;   // higher 16 bits of address
} __attribute__((packed));
```

und es gibt (wie bei der GDT) zusätzlich einen Pointer auf den Anfang der Tabelle:

19e ⟨*type definitions* 2d⟩+≡ (2a) ◁19d 22a▷

```
struct idt_ptr {
    unsigned int limit   : 16;
    unsigned int base    : 32;
} __attribute__((packed));
```

Wir verwenden zwei globale Variablen für die Tabelle und den Pointer:

20a ⟨*global variables* 2c⟩+≡ (2a) ◁16a 23c▷

```
struct idt_entry idt[256] = { 0 };
struct idt_ptr idtp;
```

Wir können einen Deskriptor mit

20b ⟨*function prototypes* 3b⟩+≡ (2a) ◁18b 20d▷

```
void fill_idt_entry (unsigned char num, unsigned long address,
    unsigned short gdtsel, unsigned char flags, unsigned char type);
```

Benutzt `fill_idt_entry` 20c.

füllen:

20c ⟨*function implementations* 3e⟩+≡ (2a) ◁18c 21b▷

```
void fill_idt_entry (unsigned char num, unsigned long address,
    unsigned short gdtsel, unsigned char flags, unsigned char type) {
  if (num >= 0 && num < 256) {
    idt[num].addr_low  = address & 0xFFFF; // address is the handler address
    idt[num].addr_high = (address >> 16) & 0xFFFF;
    idt[num].gdtsel    = gdtsel;           // GDT sel.: user mode or kernel mode?
    idt[num].zeroes    = 0;
    idt[num].flags     = flags;
    idt[num].type      = type;
  }
}
```

Definiert:
`fill_idt_entry`, benutzt im Teils 20 und 24e.

Wir definieren Interrupt-Handler-Funktionen `irq0` bis `irq15` in der Assembler-Datei. Im C-Programm müssen wir die Funktionen als extern deklarieren:

20d ⟨*function prototypes* 3b⟩+≡ (2a) ◁20b 21a▷

```
extern void irq0(), irq1(), irq2(),  irq3(),  irq4(),  irq5(),  irq6(),  irq7();
extern void irq8(), irq9(), irq10(), irq11(), irq12(), irq13(), irq14(), irq15();
```

Dann können wir die Handler auf die einzelnen IDT-Einträge verteilen:

20e ⟨*install the interrupt handlers* 20e⟩≡ (21c)

```
⟨install the IDT 24a⟩
⟨install the fault handlers 24f⟩
⟨remap the interrupts to 32..47 19a⟩
set_irqmask (0xFFFF);           // initialize IRQ mask
enable_interrupt (IRQ_SLAVE);   // IRQ slave

// flags: 1 (present), 11 (DPL 3), 0; type: 1110 (32 bit interrupt gate)
fill_idt_entry (32, (unsigned int)irq0,  0x08, 0b1110, 0b1110);
fill_idt_entry (33, (unsigned int)irq1,  0x08, 0b1110, 0b1110);
fill_idt_entry (34, (unsigned int)irq2,  0x08, 0b1110, 0b1110);
fill_idt_entry (35, (unsigned int)irq3,  0x08, 0b1110, 0b1110);
```

```
fill_idt_entry (36, (unsigned int)irq4,  0x08, 0b1110, 0b1110);
fill_idt_entry (37, (unsigned int)irq5,  0x08, 0b1110, 0b1110);
fill_idt_entry (38, (unsigned int)irq6,  0x08, 0b1110, 0b1110);
fill_idt_entry (39, (unsigned int)irq7,  0x08, 0b1110, 0b1110);
fill_idt_entry (40, (unsigned int)irq8,  0x08, 0b1110, 0b1110);
fill_idt_entry (41, (unsigned int)irq9,  0x08, 0b1110, 0b1110);
fill_idt_entry (42, (unsigned int)irq10, 0x08, 0b1110, 0b1110);
fill_idt_entry (43, (unsigned int)irq11, 0x08, 0b1110, 0b1110);
fill_idt_entry (44, (unsigned int)irq12, 0x08, 0b1110, 0b1110);
fill_idt_entry (45, (unsigned int)irq13, 0x08, 0b1110, 0b1110);
fill_idt_entry (46, (unsigned int)irq14, 0x08, 0b1110, 0b1110);
fill_idt_entry (47, (unsigned int)irq15, 0x08, 0b1110, 0b1110);
```

Benutzt `enable_interrupt` 21b, `fill_idt_entry` 20c, `IRQ_SLAVE` 18a, und `set_irqmask` 21b.

wobei wir noch

21a ⟨*function prototypes* 3b⟩+≡ (2a) ◁20d 23d▷

```
static void set_irqmask (unsigned short mask);
static void enable_interrupt (int number);
unsigned short get_irqmask ();
```

Benutzt `enable_interrupt` 21b, `get_irqmask` 21b, und `set_irqmask` 21b.

implementieren:

21b ⟨*function implementations* 3e⟩+≡ (2a) ◁20c 23a▷

```
static void set_irqmask (unsigned short mask) {
  outportb (IO_PIC_MASTER_DATA, (char)(mask % 256) );
  outportb (IO_PIC_SLAVE_DATA,  (char)(mask >> 8)  );
}

unsigned short get_irqmask () {
  return inportb (IO_PIC_MASTER_DATA)
      + (inportb (IO_PIC_SLAVE_DATA) << 8);
}

static void enable_interrupt (int number) {
  set_irqmask (
    get_irqmask ()        // the current value
    & ~(1 << number)      // 16 one-bits, but bit "number" cleared
  );
}
```

Definiert:
`enable_interrupt`, benutzt im Teils 20e und 21a.
`get_irqmask`, benutzt im Teil 21a.
`set_irqmask`, benutzt im Teils 20e und 21a.
Benutzt `inportb` 18c, `IO_PIC_MASTER_DATA` 18d, `IO_PIC_SLAVE_DATA` 18d, und `outportb` 18c.

Das Installieren der Interrupt Handler im Code Chunk ⟨*install the interrupt handlers* 20e⟩ muss bei der Systeminitialisierung erfolgen, also:

21c ⟨*kernel main: initialize system* 21c⟩≡ (2b) 27b▷

```
⟨install the interrupt handlers 20e⟩
```

Unsere Interrupt-Handler erhalten die Registerinhalte, dafür legen wir einen eigenen Datentyp an:

22a ⟨*type definitions* 2d⟩+≡ (2a) ◁19e

```
typedef struct {
  unsigned int gs, fs, es, ds;
  unsigned int edi, esi, ebp, esp, ebx, edx, ecx, eax;
  unsigned int int_no, err_code;
  unsigned int eip, cs, eflags, useresp, ss;
} context_t;
```

In der Assembler-Datei liegen die Anfangsstücke der Interrupt-Handler:

22b ⟨*start.asm* 22b⟩≡ 24c▷

```
global irq0, irq1, irq2,  irq3,  irq4,  irq5,  irq6,  irq7
global irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15

%macro irq_macro 1
       cli                  ; disable interrupts
       push byte 0          ; error code (none)
       push byte %1         ; interrupt number
       jmp irq_common_stub  ; rest is identical for all handlers
%endmacro

irq0:  irq_macro 32
irq1:  irq_macro 33
irq2:  irq_macro 34
irq3:  irq_macro 35
irq4:  irq_macro 36
irq5:  irq_macro 37
irq6:  irq_macro 38
irq7:  irq_macro 39
irq8:  irq_macro 40
irq9:  irq_macro 41
irq10: irq_macro 42
irq11: irq_macro 43
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
irq15: irq_macro 47

extern irq_handler          ; defined in the C source file

irq_common_stub:            ; this is the identical part
       pusha
       push ds
       push es
       push fs
       push gs
       push esp  ; pointer to the context_t
       call irq_handler     ; call C function
       pop esp
       pop gs
       pop fs
       pop es
       pop ds
       popa
       add esp, 8
```

```
        iret
```
Benutzt `irq_handler` 23a.

– wir haben in der Vorlesung besprochen, warum die Register in dieser Reihenfolge auf den Stack gelegt (und später wieder ausgelesen) werden.

(*Achtung: Sie können diesen Code Chunk nicht in dieser NoWeb-Datei verändern; er wird nicht exportiert.*)

Es fehlt nun noch die C-Funktion `irq_handler()`:

23a ⟨*function implementations* 3e⟩+≡ (2a) ◁21b 23e▷
```
void irq_handler (context_t *r) {
  int number = r->int_no - 32;                    // interrupt number
  void (*handler)(context_t *r);                  // type of handler functions

  if (number >= 8)
    outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT);  // notify slave  PIC
  outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);   // notify master PIC

  handler = interrupt_handlers[number];
  if (handler != NULL)  handler (r);
}
```
Definiert:
`irq_handler`, benutzt im Teil 22b.
Benutzt `interrupt_handlers` 23c, `IO_PIC_MASTER_CMD` 18d, `IO_PIC_SLAVE_CMD` 18d, und `outportb` 18c.

Sie verwendet die Konstante

23b ⟨*constants* 7b⟩+≡ (2a) ◁18d
```
#define END_OF_INTERRUPT  0x20
```

und das Array

23c ⟨*global variables* 2c⟩+≡ (2a) ◁20a 25b▷
```
void *interrupt_handlers[16] = { 0 };
```
Definiert:
`interrupt_handlers`, benutzt im Teil 23.

Um einen Interrupt-Handler zu installieren (also seine Adresse in das Array einzutragen), verwenden wir

23d ⟨*function prototypes* 3b⟩+≡ (2a) ◁21a 24b▷
```
void install_interrupt_handler (int irq, void (*handler)(context_t *r));
```
Benutzt `install_interrupt_handler` 23e.

mit folgender Implementierung:

23e ⟨*function implementations* 3e⟩+≡ (2a) ◁23a 25d▷
```
void install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
  if (irq >= 0 && irq < 16)
    interrupt_handlers[irq] = handler;
}
```
Definiert:
`install_interrupt_handler`, benutzt im Teil 23d.
Benutzt `interrupt_handlers` 23c.

Bei der Initialisierung des Kernels müssen wir noch das IDTR-Register laden:

24a ⟨*install the IDT* 24a⟩≡ (20e)
```
idtp.limit = (sizeof (struct idt_entry) * 256) - 1;   // must do -1
idtp.base  = (int) &idt;
idt_load ();
```

wobei der letzte Befehl wieder als Assembler-Code vorliegt:

24b ⟨*function prototypes* 3b⟩+≡ (2a) ◁23d 24d▷
```
extern void idt_load ();
```

24c ⟨*start.asm* 22b⟩+≡ ◁22b 25a▷
```
extern idtp  ; defined in the C file
global idt_load
idt_load:    lidt [idtp]
             ret
```

8 Faults

Das Fault-Handling funktioniert ganz ähnlich wie die Interrupt-Behandlung, darum gehen wir hier auf die Details nur kurz ein.

Die Funktionen `isr0()`, ..., `isr31()` implementieren wir wieder als Assembler-Funktionen; im C-Programm müssen wir ihre Namen bekannt machen:

24d ⟨*function prototypes* 3b⟩+≡ (2a) ◁24b 25c▷
```
extern void isr0(),  isr1(),  isr2(),  isr3(),  isr4(),  isr5(),
   isr6(),  isr7(),  isr8(),  isr9(),  isr10(), isr11(), isr12(),
   isr13(), isr14(), isr15(), isr16(), isr17(), isr18(), isr19(),
   isr20(), isr21(), isr22(), isr23(), isr24(), isr25(), isr26(),
   isr27(), isr28(), isr29(), isr30(), isr31();
```

Um die IDT-Einträge für die 32 Fault-Handler anzulegen, benutzen wir ein kleines Makro:

24e ⟨*macros* 7a⟩+≡ (2a) ◁17a
```
#define FILL_IDT(i) \
  fill_idt_entry (i, (unsigned int)isr##i, 0x08, 0b1110, 0b1110)
```
Definiert:
`FILL_IDT`, benutzt im Teil 24f.
Benutzt `fill_idt_entry` 20c.

und können dann mit wenigen Zeilen die 32 Eintragungen vornehmen:

24f ⟨*install the fault handlers* 24f⟩≡ (20e)
```
FILL_IDT( 0); FILL_IDT( 1); FILL_IDT( 2); FILL_IDT( 3); FILL_IDT( 4);
FILL_IDT( 5); FILL_IDT( 6); FILL_IDT( 7); FILL_IDT( 8); FILL_IDT( 9);
FILL_IDT(10); FILL_IDT(11); FILL_IDT(12); FILL_IDT(13); FILL_IDT(14);
FILL_IDT(15); FILL_IDT(16); FILL_IDT(17); FILL_IDT(18); FILL_IDT(19);
FILL_IDT(20); FILL_IDT(21); FILL_IDT(22); FILL_IDT(23); FILL_IDT(24);
FILL_IDT(25); FILL_IDT(26); FILL_IDT(27); FILL_IDT(28); FILL_IDT(29);
FILL_IDT(30); FILL_IDT(31);
```
Benutzt `FILL_IDT` 24e.

In der Assembler-Datei geben wir an, dass die Symbole exportiert werden sollen:

25a ⟨*start.asm* 22b⟩+≡ ◁24c

```
global isr0,  isr1,  isr2,  isr3,  isr4,  isr5,  isr6,  isr7,  isr8
global isr9,  isr10, isr11, isr12, isr13, isr14, isr15, isr16, isr17
global isr18, isr19, isr20, isr21, isr22, isr23, isr24, isr25, isr26
global isr27, isr28, isr29, isr30, isr31
```

Dann brauchen wir ein Array mit Fehlermeldungen; hinter Fault-Nummer 18 kommt nichts mehr: Die restlichen Werte sind reserviert.

25b ⟨*global variables* 2c⟩+≡ (2a) ◁23c 27d▷

```
char *exception_messages[] = {
  "Division By Zero",       "Debug",                          //  0,  1
  "Non Maskable Interrupt", "Breakpoint",                     //  2,  3
  "Into Detected Overflow", "Out of Bounds",                  //  4,  5
  "Invalid Opcode",         "No Coprocessor",                 //  6,  7
  "Double Fault",           "Coprocessor Segment Overrun",    //  8,  9
  "Bad TSS",                "Segment Not Present",            // 10, 11
  "Stack Fault",            "General Protection Fault",       // 12, 13
  "Page Fault",             "Unknown Interrupt",              // 14, 15
  "Coprocessor Fault",      "Alignment Check",                // 16, 17
  "Machine Check",                                            // 18
  "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
  "Reserved", "Reserved", "Reserved", "Reserved", "Reserved",
  "Reserved", "Reserved", "Reserved"                          // 19..31
};
```

Definiert:
`exception_messages`, benutzt im Teil 25e.

Den eigentlichen Fault-Handler

25c ⟨*function prototypes* 3b⟩+≡ (2a) ◁24d 27c▷

```
void fault_handler (context_t *r);
```

Benutzt `fault_handler` 25d.

präsentieren wir hier in der vereinfachten Variante (die noch keine Prozesse berücksichtigt). Der Handler gibt einige Informationen aus und hält dann das System an.

25d ⟨*function implementations* 3e⟩+≡ (2a) ◁23e 27e▷

```
void fault_handler (context_t *r) {
  if (r->int_no >= 0 && r->int_no < 32) {
    ⟨fault handler: display status information 25e⟩
    printf ("System Stops\n");
    asm ("cli; \n hlt;");
  }
}
```

Definiert:
`fault_handler`, benutzt im Teil 25c.

In der Ausgabe erscheinen Name und Nummer des Faults sowie die Inhalte einiger Register.

25e ⟨*fault handler: display status information* 25e⟩≡ (25d)

```
printf ("'%s' (%d) Exception at 0x%08x.\n",
  exception_messages[r->int_no], r->int_no, r->eip);
```

```
  printf ("eflags: 0x%08x  errcode: 0x%08x\n", r->eflags, r->err_code);
  printf ("eax: %08x  ebx: %08x  ecx: %08x  edx: %08x \n",
    r->eax, r->ebx, r->ecx, r->edx);
  printf ("eip: %08x  esp: %08x  int: %8d  err: %8d \n",
    r->eip, r->esp, r->int_no, r->err_code);
  printf ("ebp: %08x  cs: %d  ds: %d  es: %d  fs: %d  ss: %x \n",
    r->ebp, r->cs, r->ds, r->es, r->fs, r->ss);
```

Benutzt `exception_messages` 25b.

9 Tastatur-Treiber

Ab hier geben Sie die Lösungen zu den Aufgaben aus Übungsblatt 9 ein. Wollen Sie Code zwischendurch auskommentieren, können Sie z. B. einfach einen Chunk-Namen verwenden, der nicht benutzt wird.

27a ⟨*nonsense chunk* 27a⟩≡
```
// ich werde nicht benutzt.
```

Hier können Sie Code ergänzen, der bei der Initialisierung des Kernels ausgeführt werden soll (und diesen Kommentar danach löschen):

27b ⟨*kernel main: initialize system* 21c⟩+≡ (2b) ◁21c
```
// Platz fuer Code
```

27c ⟨*function prototypes* 3b⟩+≡ (2a) ◁25c
```
// Platz fuer Prototypen
```

27d ⟨*global variables* 2c⟩+≡ (2a) ◁25b
```
// Platz fuer Deklaration globaler Variablen
```

27e ⟨*function implementations* 3e⟩+≡ (2a) ◁25d
```
// Platz fuer Implementierungen
```

Der folgende Code Chunk wird aus der `main()`-Funktion heraus aufgerufen:

27f ⟨*kernel main: user-defined tests* 27f⟩≡ (2b)
```
printf ("Hier koennte Ihr Test stehen.\n");
```

Wichtig: Bitte tragen Sie Code nicht einfach komplett in die oben vorbereiteten Code Chunks ein – sie stehen nur hier, damit Sie auf Anhieb sehen, welche Chunks vom Rest des Kernel-Codes eingebunden werden; für die Musterlösung habe ich diese Chunks verwendet.

Sie schreiben ein Literate Program, sollten also eine Geschichte erzählen. Grob ist sie ja schon durch die Aufgabenstellungen vorgegeben.