

Mini-Ulix: Übungsblatt 5

Hans-Georg Eßer

18. November 2013

Inhaltsverzeichnis

1	Kernel-Layout	3
2	Speicher einrichten: Die GDT	3
3	Paging	5
3.1	Page Directory und Page Table	5
3.2	Funktionen für den Zugriff	6
3.3	Initialisierung des Paging	8
4	Verwaltung der Rahmen	8
4.1	Datenstrukturen	9
4.2	Funktionen für den Zugriff	9
5	Verwaltung der Seiten	11
6	Hauptprogramm: Tests	14
6.1	Das System initialisieren	14
6.2	Die Tests	15
7	Hilfsfunktionen	16
7.1	Speicher initialisieren mit <code>memset</code>	16
7.2	Text mit <code>printf</code> ausgeben	16
7.3	Den Bildschirm mit <code>clrscr</code> löschen	17
7.4	String kopieren mit <code>strncpy</code>	18
7.5	Hexdump	18
8	Interrupts	19
9	Faults	25

1 Kernel-Layout

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

3a `<ulix.c 3a>≡`
 `<constants 7c>`
 `<type definitions 3b>`
 `<global variables 3c>`
 `<macros 7b>`
 `<function prototypes 4a>`
 `<function implementations 4d>`
 `<kernel main 14a>`

Auf einen Abdruck der Assembler-Datei `start.asm` verzichten wir in dieser Ausarbeitung; sie wurde gegenüber früheren Tutorials nicht verändert. Behalten Sie aber im Hinterkopf, dass die eigentliche Initialisierung des System dort im Abschnitt `start` mit dem Einrichten der Trick-GDT beginnt.

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`).

3b `<type definitions 3b>≡` (3a) 5a>
 `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:

3c `<global variables 3c>≡` (3a) 6a>
 `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:

4a *<function prototypes 4a>*≡ (3a) 4b>
`extern void gdt_flush();`

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

4b *<function prototypes 4a>*+≡ (3a) <4a 4c>
`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

4c *<function prototypes 4a>*+≡ (3a) <4b 6b>
`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:

4d *<function implementations 4d>*≡ (3a) 7a>
`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:

4e *<kernel main: install GDT 4e>*≡ (14a)
`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:

5a (3a) <3b 5b>
<type definitions 3b>+≡
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 6a.

[page_table_desc](#), benutzt im Teils 6b und 7a.

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

5b (3a) <5a 11a>
<type definitions 3b>+≡
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 Teils 6c und 7a.

[page_table](#), benutzt im Teils 6a und 11–13.

- 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:

```
6a  <global variables 3c>+≡ (3a) <3c 9a>
    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 8, 11c, 13, und 15a.

current_pt, benutzt im Teil 8.

kernel_pd, nicht benutzt.

kernel_pt, nicht benutzt.

Benutzt page_directory 5a und page_table 5b.

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

```
6b  <function prototypes 4a>+≡ (3a) <4c 6c>
    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 5a.

und

```
6c  <function prototypes 4a>+≡ (3a) <6b 9d>
    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 5b.

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:

```
7a  <function implementations 4d>+= (3a) <4d 10a>
    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 5b und page_table_desc 5a.

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

```
7b  <macros 7b>+= (3a) 9c>
    #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:

```
7c  <constants 7c>+= (3a) 9b>
    #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 0xC0000000 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:

```
8a  <kernel main: setup paging 8a>≡ (14a) 8b>
    for (int i=1; i<1024; i++) {
        fill_page_table_desc (&(current_pd->ptds[i]), false, false, false, 0);
    };
```

Benutzt `current_pd` 6a.

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:

```
8b  <kernel main: setup paging 8a>+≡ (14a) <8a 8c>
    KMAPD ( &(current_pd->ptds[ 0]), (unsigned int)(current_pt)-0xC0000000 );
    KMAPD ( &(current_pd->ptds[768]), (unsigned int)(current_pt)-0xC0000000 );
```

Benutzt `current_pd` 6a und `current_pt` 6a.

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

```
8c  <kernel main: setup paging 8a>+≡ (14a) <8b 8d>
    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` 6a.

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

```
8d  <kernel main: setup paging 8a>+≡ (14a) <8c>
    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` 6a.

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 Datenstrukturen

Wir legen eine Rahmentabelle `fable` 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`:

```
9a <global variables 3c>+≡ (3a) <6a 14b>
    unsigned int free_frames = NUMBER_OF_FRAMES;
    char place_for_fable[NUMBER_OF_FRAMES/8];
    unsigned int* ftable = (unsigned int*)&place_for_fable;
```

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`:

```
9b <constants 7c>+≡ (3a) <7c 19b>
    #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 `fable` zugreifen zu können, definieren wir zunächst zwei Helfer-Makros:

```
9c <macros 7b>+≡ (3a) <7b 12a>
    #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

```
9d <function prototypes 4a>+≡ (3a) <6c 10b>
    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

```
9e <calculate index and offset 9e>≡ (10a)
    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:

10a *<function implementations 4d>+≡* (3a) <7a 10c>

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

static void clear_frame (unsigned int frame_addr) {
    <calculate index and offset 9e>
    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 9e>
    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

10b *<function prototypes 4a>+≡* (3a) <9d 11b>

```
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:

10c *<function implementations 4d>+≡* (3a) <10a 11c>

```
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:

11a *<type definitions 3b>+≡* (3a) *<5b 21b>*

```

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

11b *<function prototypes 4a>+≡* (3a) *<10b 12b>*

```

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):

11c *<function implementations 4d>+≡* (3a) *<10c 12c>*

```

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` 6a und `page_table` 5b.

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:

12a *<macros 7b>+≡* (3a) *<9c 19a>*

```
#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

12b *<function prototypes 4a>+≡* (3a) *<11b 13c>*

```
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:

12c *<function implementations 4d>+≡* (3a) *<11c 13d>*

```
unsigned int* request_new_page (int need_more_pages) {
    <page request implementation 12d>
}
```

12d *<page request implementation 12d>≡* (12c) *<12e>*

```
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:

12e *<page request implementation 12d>+≡* (12c) *<12d 13a>*

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

Benutzt `page_table 5b`.

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 Seiten-

tabelle verwenden wir dann den bereits angeforderten Frame und benötigen danach einen neuen.

13a *<page request implementation 12d>+≡* (12c) <12e 13b>

```

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 6a und page_table 5b.

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:

13b *<page request implementation 12d>+≡* (12c) <13a>

```

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 6a und page_table 5b.

(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

13c *<function prototypes 4a>+≡* (3a) <12b 16a>

```

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.

13d *<function implementations 4d>+≡* (3a) <12c 16b>

```

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 (framenos<<12); // expects an address, not an ID
    asm volatile ("invlpg %0" : : "m"(*(char*)(pagenos<<12)) );
    // gdt_flush ();
};

```

Benutzt `current_pd` 6a und `page_table` 5b.

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

6 Hauptprogramm: Tests

Jetzt bleibt nur noch ein Test der neuen Features. Den bauen wir in das Hauptprogramm ein, das zunächst die Segmentierung und das Paging initialisiert und dann einige Tests der Frame- und Seitenverwaltung durchführt.

6.1 Das System initialisieren

Die `main`-Funktion hat den folgenden Aufbau:

```

14a  <kernel main 14a>≡ (3a)
    int main () {
        <kernel main: initialize variables 17a>
        printf ("[1] entering main()\n");

        <kernel main: setup paging 8a>
        <kernel main: install GDT 4e> // replace "trick GDT" with regular GDT

        paging_ready = true;
        printf ("[4] regular GDT is active\n");

        <kernel main: map physical RAM 15a>
        <kernel main: setup frame table 15b>
        <kernel main: initialize system 23b>
        <kernel main: feature tests 15c>
        <kernel main: user-defined tests 28f>
        for (;;) // infinite loop
    }

```

Hier setzen wir auch die Variable `paging_ready`, die nur von `kputch` ausgewertet wird; die Funktion findet darüber heraus, an welcher Adresse sie den Textmodus-Framebuffer der Grafikkarte findet. Wir müssen sie noch deklarieren, anfangs ist sie `false`:

```

14b  <global variables 3c>+≡ (3a) <9a 18c>
    int paging_ready = false;

```

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

```
15a <kernel main: map physical RAM 15a>≡ (14a)
    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 6a.
```

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.

```
15b <kernel main: setup frame table 15b>≡ (14a)
    memset (ftable, 0, NUMBER_OF_FRAMES/8); // all frames are free
    memset (ftable, 0xff, 128);
    free_frames -= 1024;
```

6.2 Die Tests

Es bleiben nur noch die Tests. Wir überprüfen die Funktionen `request_new_frame`, `release_frame`, `request_new_page` und `release_page`:

```
15c <kernel main: feature tests 15c>≡ (14a)
    /*
    printf ("TEST req_frame: free_frames = %d, ", free_frames);
    int fid = request_new_frame ();
    printf ("frameid = 0x%x, free_frames = %d\n", fid, free_frames);

    printf ("TEST req_page: free_frames = %d, ", free_frames);
    unsigned int *address = request_new_page (0);
    printf ("addr = 0x%x, free_frames = %d\n", address, free_frames);

    // Use new page for a string
    memset (address, 'z', PAGE_SIZE);
    char *string = (char *)address; string[10] = 0;
    printf ("Test-String of 10 'z's: %s -- address: 0x%x\n",
        string, (unsigned int)string);
    printf ("pageno_to_frameno (0x%x) = 0x%x\n",
```

```

        (unsigned int)address >> 12,
        pageno_to_frameno ((unsigned int)address >> 12));

release_page ((unsigned int)address >> 12);
printf ("After release_page (0x%x): free_frames = %d\n",
        (unsigned int)address >> 12, free_frames);
printf ("pageno_to_frameno (0x%x) = %d (-1: not mapped)\n",
        (unsigned int)address >> 12,
        pageno_to_frameno ((unsigned int)address >> 12));
*/

```

Die letzte (auskommentierte) Codezeile würde versuchen, auf die freigegebene Seite zuzugreifen – wenn Sie diese Zeile aktivieren, hängt sich das System auf, was ein korrektes Verhalten ist, da dieses Mini-Ulix noch keinen Fault-Handler besitzt.

7 Hilfsfunktionen

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

7.1 Speicher initialisieren mit memset

Die Funktionen

16a *<function prototypes 4a>+≡* (3a) <13c 16c>

```

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:

16b *<function implementations 4d>+≡* (3a) <13d 17c>

```

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;
}

```

7.2 Text mit printf ausgeben

Die Funktion

16c *<function prototypes 4a>+≡* (3a) <16a 17b>

```

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:

```
17a <kernel main: initialize variables 17a>≡ (14a)
    posx = 0; posy = 8; // set cursor
```

Die `printf`-Funktion nutzt die einfachere Funktion

```
17b <function prototypes 4a>+≡ (3a) <16c 17d>
    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 Cursorposition zu merken, verwendet sie die Variablen `posx` und `posy`.

```
17c <function implementations 4d>+≡ (3a) <16b 18b>
    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:

```
17d <function prototypes 4a>+≡ (3a) <17b 18a>
    extern void uartputc (int c);
```

7.3 Den Bildschirm mit `clrscr` löschen

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

18a *<function prototypes 4a>+≡* (3a) <17d 18d>
`void clrscr ();`

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

18b *<function implementations 4d>+≡* (3a) <17c 18e>
`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:

18c *<global variables 3c>+≡* (3a) <14b 21d>
`int posx, posy;`

7.4 String kopieren mit strncpy

18d *<function prototypes 4a>+≡* (3a) <18a 18f>
`void *strncpy(void *dest, const void *src, int count);`

18e *<function implementations 4d>+≡* (3a) <18b 18g>
`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;
}`

7.5 Hexdump

Zum Bearbeiten der Übungsaufgabe stellen wir noch die folgende Funktion

18f *<function prototypes 4a>+≡* (3a) <18d 20a>
`void hexdump (unsigned int start, unsigned int end);`

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

18g *<function implementations 4d>+≡* (3a) <18e 20b>
`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:

19a `<macros 7b>+≡` (3a) <12a 26b>
`#define PEEK(addr) (*(unsigned char *)(addr))`

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

19b `<constants 7c>+≡` (3a) <9b 19c>
`#define asm __asm__`
`#define volatile __volatile__`
`#define NULL ((void*) 0)`

8 Interrupts

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

19c `<constants 7c>+≡` (3a) <19b 20c>
`#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 22b.
 IRQ_TIMER, nicht benutzt.

Weiter geht es mit den in- und out-Befehlen

20a *<function prototypes 4a>+≡* (3a) <18f 21e>
 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 20b, inportw 20b, outportb 20b, und outportw 20b.

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.

20b *<function implementations 4d>+≡* (3a) <18g 21f>
 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 20a und 23a.
 inportw, benutzt im Teil 20a.
 outportb, benutzt im Teils 20, 21a, 23a, und 24.
 outportw, benutzt im Teil 20a.

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

20c *<constants 7c>+≡* (3a) <19c 25a>
 // 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.COMMAND, benutzt im Teils 20e und 24.
 IO.PIC.MASTER.DATA, benutzt im Teils 20e, 21a, und 23a.
 IO.PIC.SLAVE.COMMAND, benutzt im Teils 20e und 24.
 IO.PIC.SLAVE.DATA, benutzt im Teils 20e, 21a, und 23a.

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

20d *<remap the interrupts to 32..47 20d>≡* (22b)
<PIC: program/initialize the PICs 20e>
<PIC: set the initial interrupt mask 21a>

20e *<PIC: program/initialize the PICs 20e>≡* (20d)
 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 20c, IO_PIC_MASTER_DATA 20c, IO_PIC_SLAVE_CMD 20c,
IO_PIC_SLAVE_DATA 20c, und outportb 20b.

```

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

```

21a <PIC: set the initial interrupt mask 21a>≡ (20d)
outportb (IO_PIC_MASTER_DATA, 0x00); // PIC1: mask 0
outportb (IO_PIC_SLAVE_DATA, 0x00); // PIC2: mask 0
Benutzt IO_PIC_MASTER_DATA 20c, IO_PIC_SLAVE_DATA 20c, und outportb 20b.

```

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

```

21b <type definitions 3b>+≡ (3a) <11a 21c>
struct idt_entry {
    unsigned int addr_low : 16; // lower 16 bits of address
    unsigned int gdtssel : 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:

```

21c <type definitions 3b>+≡ (3a) <21b 23c>
struct idt_ptr {
    unsigned int limit : 16;
    unsigned int base : 32;
} __attribute__((packed));

```

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

```

21d <global variables 3c>+≡ (3a) <18c 25b>
struct idt_entry idt[256] = { 0 };
struct idt_ptr idtp;

```

Wir können einen Deskriptor mit

```

21e <function prototypes 4a>+≡ (3a) <20a 22a>
void fill_idt_entry (unsigned char num, unsigned long address,
    unsigned short gdtssel, unsigned char flags, unsigned char type);
Benutzt fill_idt_entry 21f.

```

füllen:

```

21f <function implementations 4d>+≡ (3a) <20b 23a>
void fill_idt_entry (unsigned char num, unsigned long address,

```

```

        unsigned short gdtssel, 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].gdtssel   = gdtssel;           // 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 21e, 22b, und 26b.

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

22a *<function prototypes 4a>+≡* (3a) <21e 22c>
 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:

22b *<install the interrupt handlers 22b>≡* (23b)
<install the IDT 25e>
<install the fault handlers 26c>
<remap the interrupts to 32..47 20d>
 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 23a, fill_idt_entry 21f, IRQ_SLAVE 19c, und set_irqmask 23a.

wobei wir noch

22c *<function prototypes 4a>+≡* (3a) <22a 25c>
 static void set_irqmask (unsigned short mask);
 static void enable_interrupt (int number);
 unsigned short get_irqmask ();

Benutzt enable_interrupt 23a, get_irqmask 23a, und set_irqmask 23a.

implementieren:

```
23a  <function implementations 4d>+≡ (3a) <21f 24>
    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 Teil 22.
get_irqmask, benutzt im Teil 22c.
set_irqmask, benutzt im Teil 22.

Benutzt inportb 20b, IO_PIC_MASTER_DATA 20c, IO_PIC_SLAVE_DATA 20c, und outportb 20b.

Das Installieren der Interrupt Handler im Code Chunk *<install the interrupt handlers 22b>* muss bei der Systeminitialisierung erfolgen, also:

```
23b  <kernel main: initialize system 23b>≡ (14a) 28b>
    <install the interrupt handlers 22b>
```

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

```
23c  <type definitions 3b>+≡ (3a) <21c
    struct regs {
        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;
    };
}
```

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

```
23d  <start.asm 23d>≡ 25g>
    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 struct regs
    call irq_handler ; call C function
    pop esp
    pop gs
    pop fs
    pop es
    pop ds
    popa
    add esp, 8
    iret

```

Benutzt irq_handler 24.

– 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()`:

```

24  <function implementations 4d>+≡ (3a) <23a 25d>
    void irq_handler (struct regs *r) {
        int number = r->int_no - 32; // interrupt number
        void (*handler)(struct regs *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 23d.

Benutzt `interrupt_handlers` 25b, `IO_PIC_MASTER_CMD` 20c, `IO_PIC_SLAVE_CMD` 20c,
und `outportb` 20b.

Sie verwendet die Konstante

25a *<constants 7c>+≡* (3a) *<20c*
`#define END_OF_INTERRUPT 0x20`

und das Array

25b *<global variables 3c>+≡* (3a) *<21d 26e>*
`void *interrupt_handlers[16] = { 0 };`

Definiert:

`interrupt_handlers`, benutzt im Teils 24 und 25d.

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

25c *<function prototypes 4a>+≡* (3a) *<22c 25f>*
`void install_interrupt_handler (int irq, void (*handler)(struct regs *r));`

Benutzt `install_interrupt_handler` 25d.

mit folgender Implementierung:

25d *<function implementations 4d>+≡* (3a) *<24 27b>*
`void install_interrupt_handler (int irq, void (*handler)(struct regs *r)) {
 if (irq >= 0 && irq < 16)
 interrupt_handlers[irq] = handler;
}`

Definiert:

`install_interrupt_handler`, benutzt im Teil 25c.

Benutzt `interrupt_handlers` 25b.

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

25e *<install the IDT 25e>≡* (22b)
`idt.limit = (sizeof (struct idt_entry) * 256) - 1; // must do -1
idt.base = (int) &idt;
idt_load ();`

wobei der letzte Befehl wieder als Assembler-Code vorliegt:

25f *<function prototypes 4a>+≡* (3a) *<25c 26a>*
`extern void idt_load ();`

25g *<start.asm 23d>+≡* *<23d 26d>*
`extern idtp ; defined in the C file
global idt_load
idt_load: lidt [idt]
 ret`

9 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:

```
26a  <function prototypes 4a>+≡ (3a) <25f 27a>
      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:

```
26b  <macros 7b>+≡ (3a) <19a>
      #define FILL_IDT(i) \
            fill_idt_entry (i, (unsigned int)isr##i, 0x08, 0b1110, 0b1110)
```

Definiert:

`FILL_IDT`, benutzt im Teil 26c.

Benutzt `fill_idt_entry` 21f.

und können dann mit wenigen Zeilen die 32 Eintragungen vornehmen:

```
26c  <install the fault handlers 26c>≡ (22b)
      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` 26b.

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

```
26d  <start.asm 23d>+≡ <25g>
      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.

```
26e  <global variables 3c>+≡ (3a) <25b 28d>
      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" // 19..31
};
```

Definiert:

`exception_messages`, benutzt im Teil 27c.

Den eigentlichen Fault-Handler

```
27a <function prototypes 4a>+≡ (3a) <26a 28c>
void fault_handler (struct regs *r);
```

Benutzt `fault_handler` 27b.

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

```
27b <function implementations 4d>+≡ (3a) <25d 28e>
void fault_handler (struct regs *r) {
    if (r->int_no >= 0 && r->int_no < 32) {
        <fault handler: display status information 27c>
        printf ("System Stops\n");
        asm ("cli; \n hlt;");
    }
}
```

Definiert:

`fault_handler`, benutzt im Teil 27a.

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

```
27c <fault handler: display status information 27c>≡ (27b)
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` 26e.

10 Tastatur-Treiber

Ab hier geben Sie die Lösungen zu den Aufgaben aus Übung 5 ein. Wollen Sie Code zwischendurch auskommentieren, können Sie z. B. einfach einen Chunk-Namen verwenden, der nicht benutzt wird.

28a `<nonsense chunk 28a>≡
// ich werde nicht benutzt.`

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

28b `<kernel main: initialize system 23b>+≡ (14a) <23b
// Platz fuer Code`

28c `<function prototypes 4a>+≡ (3a) <27a
// Platz fuer Prototypen`

28d `<global variables 3c>+≡ (3a) <26e
// Platz fuer Deklaration globaler Variablen`

28e `<function implementations 4d>+≡ (3a) <27b
// Platz fuer Implementierungen`

Der letzte Code Chunk wird aus der `main()`-Funktion heraus aufgerufen.

28f `<kernel main: user-defined tests 28f>≡ (14a)
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.