61 lines added. 67 lines deleted.

```
#define false 0
#define true 1
#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
#define asm __asm__
#define volatile __volatile__
#define NULL ((void*) 0)
```

```
#define false 0
#define true 1
#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
#define asm __asm__
#define volatile __volatile__
#define NULL ((void*) 0)

// NEU
#define GET_frame_addr(x) (x & 0xFFFFF000)
#define FLAG_PRESENT          1<<0     // Bit 0: present
#define GET_present(x)    ((x & FLAG_PRESENT) != 0)
#define FLAG_WRITEABLE        1<<1     // Bit 1: writeable
#define FLAG_USER_ACCESSIBLE  1<<2     // Bit 2: user-accessible
#define FLAG_DIRTY            1<<6     // Bit 6: dirty
```

```
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));
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;
```

```
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));
```



```

typedef struct { page_table_desc ptds[1024]; } page_directory;
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;
typedef unsigned int boolean;
struct gdt_entry gdt[6];
struct gdt_ptr gp;
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;
unsigned int free_frames = NUMBER_OF_FRAMES;
char place_for_ftable[NUMBER_OF_FRAMES/8];
unsigned int* ftable = (unsigned int*)(&place_for_ftable);
int paging_ready = false;
int posx, posy;
#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)
#define INDEX_FROM_BIT(b) (b/32)   // 32 bits in an unsigned int
#define OFFSET_FROM_BIT(b) (b%32)
#define PHYSICAL(x) ((x)+0xd0000000)
#define PEEK(addr) (*(unsigned char *)(addr))
```

```

typedef unsigned int page_table_desc;  // NEU!
typedef unsigned int page_desc;        // NEU!

typedef unsigned int boolean;
struct gdt_entry gdt[6];
struct gdt_ptr gp;

// NEU:
page_table_desc kernel_pd[1024]  __attribute__ ((aligned (4096)));
page_desc kernel_pt[1024]        __attribute__ ((aligned (4096)));
page_desc kernel_pt_ram[16][1024] __attribute__ ((aligned (4096)));

// NEU:
page_table_desc *current_pd = kernel_pd;
page_desc       *current_pt = kernel_pt;
unsigned int free_frames = NUMBER_OF_FRAMES;
char place_for_ftable[NUMBER_OF_FRAMES/8];
unsigned int* ftable = (unsigned int*)(&place_for_ftable);
int paging_ready = false;
int posx, posy;
#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)
#define INDEX_FROM_BIT(b) (b/32)   // 32 bits in an unsigned int
#define OFFSET_FROM_BIT(b) (b%32)
#define PHYSICAL(x) ((x)+0xd0000000)
#define PEEK(addr) (*(unsigned char *)(addr))
```

```
74 extern void gdt_flush();
75 void gdt_set_gate (int num, unsigned long base,
76   unsigned long limit, unsigned char access, unsigned char gran);
77 void gdt_install ();
78 page_table_desc* fill_page_table_desc (page_table_desc *ptd,
79   unsigned int present, unsigned int writeable,
80   unsigned int user_accessible, unsigned int frame_addr);
81 page_desc* fill_page_desc (page_desc *pd, unsigned int present,
82   unsigned int writeable, unsigned int user_accessible,
83   unsigned int dirty, unsigned int frame_addr);
84 static void set_frame (unsigned int frame_addr);
85 static void clear_frame (unsigned int frame_addr);
86 static unsigned int test_frame (unsigned int frame_addr);
87 int request_new_frame ();
88 void release_frame (unsigned int frameaddr);
89 unsigned int pageno_to_frameno (unsigned int pageno);
90 unsigned int* request_new_page (int need_more_pages);
91 void release_page (unsigned int pageno);
92 void *memset (void *dest, char val, int count);
93 extern int printf(const char *format, ...);
94 void kputch (char c);
95 extern void uartputc (int c);
96 void clrscr ();
97 void hexdump (unsigned int start, unsigned int end);
98 void gdt_set_gate(int num, unsigned long base, unsigned long
   limit,
99                   unsigned char access, unsigned char gran) {
100   /* Setup the descriptor base address */
101   gdt[num].base_low = (base & 0xFFFF);            // 16 bits
102   gdt[num].base_middle = (base >> 16) & 0xFF;     //  8 bits
103   gdt[num].base_high = (base >> 24) & 0xFF;       //  8 bits
104
105   /* Setup the descriptor limits */
106   gdt[num].limit_low  = (limit & 0xFFFF);         // 16 bits
107   gdt[num].limit_high = ((limit >> 16) & 0x0F);   //  4 bits
108
109   /* Finally, set up the granularity and access flags */
110   gdt[num].flags = gran & 0xF;
111   gdt[num].access = access;
112 }
113
114 void gdt_install() {
115   gp.limit = (sizeof(struct gdt_entry) * 6) - 1;
116   gp.base = (int) &gdt;
117
118   gdt_set_gate(0, 0, 0, 0, 0);    // NULL descriptor
119
120   // code segment
121   gdt_set_gate(1, 0, 0xFFFFFFFF, 0b10011010, 0b1100 /* 0xCF */);
122
123   // data segment
```

```
63 extern void gdt_flush();
64 void gdt_set_gate (int num, unsigned long base,
65   unsigned long limit, unsigned char access, unsigned char gran);
66 void gdt_install ();
67 page_table_desc* fill_page_table_desc (page_table_desc *ptd,
68   unsigned int present, unsigned int writeable,
69   unsigned int user_accessible, unsigned int frame_addr);
70 page_desc* fill_page_desc (page_desc *pd, unsigned int present,
71   unsigned int writeable, unsigned int user_accessible,
72   unsigned int dirty, unsigned int frame_addr);
73 static void set_frame (unsigned int frame_addr);
74 static void clear_frame (unsigned int frame_addr);
75 static unsigned int test_frame (unsigned int frame_addr);
76 int request_new_frame ();
77 void release_frame (unsigned int frameaddr);
78 unsigned int pageno_to_frameno (unsigned int pageno);
79 unsigned int* request_new_page (int need_more_pages);
80 void release_page (unsigned int pageno);
81 void *memset (void *dest, char val, int count);
82 extern int printf(const char *format, ...);
83 void kputch (char c);
84 extern void uartputc (int c);
85 void clrscr ();
86 void hexdump (unsigned int start, unsigned int end);
87 void gdt_set_gate(int num, unsigned long base, unsigned long
   limit,
88                   unsigned char access, unsigned char gran) {
89   /* Setup the descriptor base address */
90   gdt[num].base_low = (base & 0xFFFF);            // 16 bits
91   gdt[num].base_middle = (base >> 16) & 0xFF;     //  8 bits
92   gdt[num].base_high = (base >> 24) & 0xFF;       //  8 bits
93
94   /* Setup the descriptor limits */
95   gdt[num].limit_low  = (limit & 0xFFFF);         // 16 bits
96   gdt[num].limit_high = ((limit >> 16) & 0x0F);   //  4 bits
97
98   /* Finally, set up the granularity and access flags */
99   gdt[num].flags = gran & 0xF;
100  gdt[num].access = access;
101 }
102
103 void gdt_install() {
104   gp.limit = (sizeof(struct gdt_entry) * 6) - 1;
105   gp.base = (int) &gdt;
106
107   gdt_set_gate(0, 0, 0, 0, 0);    // NULL descriptor
108
109   // code segment
110   gdt_set_gate(1, 0, 0xFFFFFFFF, 0b10011010, 0b1100 /* 0xCF */);
111
112   // data segment
```



```
124   gdt_set_gate(2, 0, 0xFFFFFFFF, 0b10010010, 0b1100 /* 0xCF */);
125
126   gdt_flush();
127 }
128 page_desc* fill_page_desc (page_desc *pd, unsigned int present,
129   unsigned int writeable, unsigned int user_accessible,
130   unsigned int dirty, unsigned int frame_addr) {
131
132   memset (pd, 0, sizeof(pd));
133
134   pd->present = present;
135   pd->writeable = writeable;
136   pd->user_accessible = user_accessible;
137   pd->dirty = dirty;
138   pd->frame_addr = frame_addr >> 12;   // right shift, 12 bits
139   return pd;
140 };
141
142 page_table_desc* fill_page_table_desc (page_table_desc *ptd,
143   unsigned int present, unsigned int writeable,
144   unsigned int user_accessible, unsigned int frame_addr) {
145
146   memset (ptd, 0, sizeof(ptd));
147
148   ptd->present = present;
149   ptd->writeable = writeable;
150   ptd->user_accessible = user_accessible;
151   ptd->frame_addr = frame_addr >> 12;   // right shift, 12 bits
152   return ptd;
153 };
154 static void set_frame (unsigned int frame_addr) {
155   unsigned int frame = frame_addr / PAGE_SIZE;
156   unsigned int index  = INDEX_FROM_BIT  (frame);
157   unsigned int offset = OFFSET_FROM_BIT (frame);
158   ftable[index] |= (1 << offset);
159 }
160
161 static void clear_frame (unsigned int frame_addr) {
162   unsigned int frame = frame_addr / PAGE_SIZE;
163   unsigned int index  = INDEX_FROM_BIT  (frame);
164   unsigned int offset = OFFSET_FROM_BIT (frame);
165   ftable[index] &= ~(1 << offset);
166 }
167
168 static unsigned int test_frame (unsigned int frame_addr) {
169   // returns true if frame is in use (false if frame is free)
170   unsigned int frame = frame_addr / PAGE_SIZE;
```

```
113   gdt_set_gate(2, 0, 0xFFFFFFFF, 0b10010010, 0b1100 /* 0xCF */);
114
115   gdt_flush();
116 }
117 page_desc* fill_page_desc (page_desc *pd, unsigned int present,
118   unsigned int writeable, unsigned int user_accessible,
119   unsigned int dirty, unsigned int frame_addr) {
120
121         // NEU:
122         unsigned int tmp = frame_addr;
123         if (present)        tmp |= FLAG_PRESENT;
124         if (writeable)      tmp |= FLAG_WRITEABLE;
125         if (user_accessible) tmp |= FLAG_USER_ACCESSIBLE;
126         if (dirty)          tmp |= FLAG_DIRTY;
127   *pd = tmp;
128
129   return pd;
130 };
131
132 page_table_desc* fill_page_table_desc (page_table_desc *ptd,
133   unsigned int present, unsigned int writeable,
134   unsigned int user_accessible, unsigned int frame_addr) {
135
136         // NEU:
137         unsigned int tmp = frame_addr;
138         if (present)        tmp |= FLAG_PRESENT;
139         if (writeable)      tmp |= FLAG_WRITEABLE;
140         if (user_accessible) tmp |= FLAG_USER_ACCESSIBLE;
141   *ptd = tmp;
142
143   return ptd;
144 };
145
146 static void set_frame (unsigned int frame_addr) {
147   unsigned int frame = frame_addr / PAGE_SIZE;
148   unsigned int index  = INDEX_FROM_BIT  (frame);
149   unsigned int offset = OFFSET_FROM_BIT (frame);
150   ftable[index] |= (1 << offset);
151 }
152
153 static void clear_frame (unsigned int frame_addr) {
154   unsigned int frame = frame_addr / PAGE_SIZE;
155   unsigned int index  = INDEX_FROM_BIT  (frame);
156   unsigned int offset = OFFSET_FROM_BIT (frame);
157   ftable[index] &= ~(1 << offset);
158 }
159
160 static unsigned int test_frame (unsigned int frame_addr) {
161   // returns true if frame is in use (false if frame is free)
162   unsigned int frame = frame_addr / PAGE_SIZE;
```

```
171   unsigned int index  = INDEX_FROM_BIT  (frame);
172   unsigned int offset = OFFSET_FROM_BIT (frame);
173   return ((ftable[index] & (1 << offset)) >> offset);
174 }
175 int request_new_frame () {
176   unsigned int frameid;
177   boolean found=false;
178   for (frameid = 0; frameid < NUMBER_OF_FRAMES; frameid++) {
179     if ( !test_frame (frameid*4096) ) {
180       found=true;
181       break;   // frame found
182     };
183   }
184   if (found) {
185     memset ((void*)PHYSICAL(frameid << 12), 0, PAGE_SIZE);
186     set_frame (frameid*4096);
187     free_frames--;
188     return frameid;
189   } else {
190     return -1;
191   }
192 };
193
194 void release_frame (unsigned int frameaddr) {
195   if ( test_frame (frameaddr) ) {
196     // only do work if frame is marked as used
197     clear_frame (frameaddr);
198     free_frames++;
199   };
200 };
201 unsigned int pageno_to_frameno (unsigned int pageno) {
202   unsigned int pdindex = pageno/1024;
203   unsigned int ptindex = pageno%1024;
204   if ( ! current_pd->ptds[pdindex].present ) {
205     return -1;       // we don't have that page table
206   } else {
207     // get the page table
208     page_table* pt = (page_table*)
209       ( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
210     if ( pt->pds[ptindex].present ) {
211       return pt->pds[ptindex].frame_addr;
212     } else {
213       return -1;     // we don't have that page
214     };
215   };
216 };
217 unsigned int* request_new_page (int need_more_pages) {
218   unsigned int newframeid = request_new_frame ();
219   if (newframeid == -1) { return NULL; }  // exit if no frame was
    found
220   unsigned int pageno = -1;
```

```
163   unsigned int index  = INDEX_FROM_BIT  (frame);
164   unsigned int offset = OFFSET_FROM_BIT (frame);
165   return ((ftable[index] & (1 << offset)) >> offset);
166 }
167 int request_new_frame () {
168   unsigned int frameid;
169   boolean found=false;
170   for (frameid = 0; frameid < NUMBER_OF_FRAMES; frameid++) {
171     if ( !test_frame (frameid*4096) ) {
172       found=true;
173       break;   // frame found
174     };
175   }
176   if (found) {
177     memset ((void*)PHYSICAL(frameid << 12), 0, PAGE_SIZE);
178     set_frame (frameid*4096);
179     free_frames--;
180     return frameid;
181   } else {
182     return -1;
183   }
184 };
185
186 void release_frame (unsigned int frameaddr) {
187   if ( test_frame (frameaddr) ) {
188     // only do work if frame is marked as used
189     clear_frame (frameaddr);
190     free_frames++;
191   };
192 };
193 unsigned int pageno_to_frameno (unsigned int pageno) {
194   unsigned int pdindex = pageno/1024;
195   unsigned int ptindex = pageno%1024;
196   if ( ! GET_present(current_pd[pdindex]) ) {
197     return -1;       // we don't have that page table
198   } else {
199     // get the page table
200     page_desc *pt = (page_desc*)
201       ( PHYSICAL(GET_frame_addr(current_pd[pdindex])) );
202     if ( GET_present (pt[ptindex]) ) {
203       return GET_frame_addr (pt[ptindex]);
204     } else {
205       return -1;     // we don't have that page
206     };
207   };
208 };
209 unsigned int* request_new_page (int need_more_pages) {
210   unsigned int newframeid = request_new_frame ();
211   if (newframeid == -1) { return NULL; }  // exit if no frame was
    found
212   unsigned int pageno = -1;
```



```
221   for (unsigned int i=0xc0000; i<1024*1024; i++) {
222     if ( pageno_to_frameno (i) == -1 ) {
223       pageno = i;
224       break;       // end loop, unmapped page was found
225     };
226   };
227
228   if ( pageno == -1 ) {
229     return NULL;   // we found no page -- whole 4 GB are mapped???
230   };
231   unsigned int pdindex = pageno/1024;
232   unsigned int ptindex = pageno%1024;
233   page_table* pt;
234   if (ptindex == 0) {
235     // last entry! // create a new page table in the reserved
    frame
236     page_table* pt = (page_table*) PHYSICAL(newframeid<<12);
237     memset (pt, 0, PAGE_SIZE);
238     KMAPD ( &(current_pd->ptds[pdindex]), newframeid << 12 );
239
240     newframeid = request_new_frame ();  // get yet another frame
241     if (newframeid == -1) {
242       return NULL;                      // exit if no frame was
    found
243       // note: we're not removing the new page table since we
    assume
244       // it will be used soon anyway
245     }
246   };
247   pt = (page_table*)(
    PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
248   // finally: enter the frame address
249   KMAP ( &(pt->pds[ptindex]), newframeid * PAGE_SIZE );
250
251   // invalidate cache entry
252   asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
253
254   memset ((unsigned int*) (pageno*4096), 0, 4096);
255   return ((unsigned int*) (pageno*4096));
256 }
257 void release_page (unsigned int pageno) {
258   int frameno = pageno_to_frameno (pageno);  // we will need this
    later
259   if ( frameno == -1 )  { return; }          // exit if no such
    page
260   unsigned int pdindex = pageno/1024;
261   unsigned int ptindex = pageno%1024;
262   page_table* pt;
263   pt = (page_table*)
264     ( PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12) );
```

```
213   for (unsigned int i=0xc0000; i<1024*1024; i++) {
214     if ( pageno_to_frameno (i) == -1 ) {
215       pageno = i;
216       break;       // end loop, unmapped page was found
217     };
218   };
219
220   if ( pageno == -1 ) {
221     return NULL;   // we found no page -- whole 4 GB are mapped???
222   };
223   unsigned int pdindex = pageno/1024;
224   unsigned int ptindex = pageno%1024;
225   page_desc *pt;
226   if (ptindex == 0) {
227     // last entry! // create a new page table in the reserved
    frame
228     page_desc *pt = (page_desc*) PHYSICAL(newframeid<<12);
229     memset (pt, 0, PAGE_SIZE);
230     KMAPD ( &(current_pd[pdindex]), newframeid << 12 );
231
232     newframeid = request_new_frame ();  // get yet another frame
233     if (newframeid == -1) {
234       return NULL;                      // exit if no frame was
    found
235       // note: we're not removing the new page table since we
    assume
236       // it will be used soon anyway
237     }
238   };
239   pt = (page_desc*)( PHYSICAL(GET_frame_addr(current_pd[pdindex]))
    );
240   // finally: enter the frame address
241   KMAP ( &(pt[ptindex]), newframeid * PAGE_SIZE );
242
243   // invalidate cache entry
244   asm volatile ("invlpg %0" : : "m"(*(char*)(pageno<<12)) );
245
246   memset ((unsigned int*) (pageno*4096), 0, 4096);
247   return ((unsigned int*) (pageno*4096));
248 }
249 void release_page (unsigned int pageno) {
250   int frameno = pageno_to_frameno (pageno);  // we will need this
    later
251   if ( frameno == -1 )  { return; }          // exit if no such
    page
252   unsigned int pdindex = pageno/1024;
253   unsigned int ptindex = pageno%1024;
254   page_desc* pt;
255   pt = (page_desc*)
256     ( PHYSICAL(GET_frame_addr(current_pd[pdindex])) );
```

```
  // 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 ();
};
void *memset (void *dest, char val, int count) {
  char *temp = (char *)dest;
  for( ; count != 0; count--) *temp++ = val;
  return dest;
}
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 Erkl%rung
  if (c == 0x100) {  //  backspace
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else uartputc(c);
}
void clrscr () {
  posx = posy = 0;
  int i;
  for ( i=0; i < 80*25; i++ ) kputch (' ');
  posx = posy = 0;
}
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));
```

```
  // write null page descriptor
  memset (&(pt[ptindex]), 0, 4);
  fill_page_desc (&(pt[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 ();
};
void *memset (void *dest, char val, int count) {
  char *temp = (char *)dest;
  for( ; count != 0; count--) *temp++ = val;
  return dest;
}
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 Erkl%rung
  if (c == 0x100) {  //  backspace
    uartputc('\b'); uartputc(' '); uartputc('\b');
  } else uartputc(c);
}
void clrscr () {
  posx = posy = 0;
  int i;
  for ( i=0; i < 80*25; i++ ) kputch (' ');
  posx = posy = 0;
}
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');
  }
}
int main () {
  posx = 0; posy = 8;  // set cursor
  printf ("[1] entering main()\n");

  for (int i=1; i<1024; i++) {
    fill_page_table_desc (&(current_pd->ptds[i]), false, false, false, 0);
  };
  KMAPD ( &(current_pd->ptds[ 0]), (unsigned int)(current_pt)-0xC0000000 );
  KMAPD ( &(current_pd->ptds[768]), (unsigned int)(current_pt)-0xC0000000 );
  for (int i=0; i<1023; i++) {
    KMAP ( &(current_pt->pds[i]), i*4096 );
  };
  printf ("[2] page directory setup, with identity mapping\n");
  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");
  gdt_install ();  // replace "trick GDT" with regular GDT

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

  memset (kernel_pt_ram, 0, 4);
```

```
      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');
  }
}
int main () {
  posx = 0; posy = 8;  // set cursor
  printf ("[1] entering main()  -- Test mit alternativen PT/PD-Typen.\n");

  for (int i=1; i<1024; i++) {
    fill_page_table_desc (&(current_pd[i]), false, false, false, 0);
  };
  KMAPD ( &(current_pd[ 0]), (unsigned int)(current_pt)-0xC0000000 );
  KMAPD ( &(current_pd[768]), (unsigned int)(current_pt)-0xC0000000 );
  for (int i=0; i<1023; i++) {
    KMAP ( &(current_pt[i]), i*4096 );
  };
  printf ("[2] page directory setup, with identity mapping\n");
  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");
  gdt_install ();  // replace "trick GDT" with regular GDT

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

  memset (kernel_pt_ram, 0, 4);
```

ORIGINAL TEXT

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


  // following line should make it hang
  // printf ("Test-String of 10 'z's: %s\n", string);
  for (;;);   // inifinite loop
}
```

CHANGED TEXT

```
  for (unsigned int fid=0; fid<NUMBER_OF_FRAMES; fid++) {
                // Index:              0..15      0..1023
    KMAP ( &(kernel_pt_ram[fid/1024][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[832+i]), physaddr );
  };

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


  // following line should make it hang
  // printf ("Test-String of 10 'z's: %s\n", string);
  for (;;);   // inifinite loop
}
```