BS-Entwicklung mit Literate Programming

Foliensatz 9

Hans-Georg Eßer
TH Nürnberg

v1.0, 02.12.2013

Chunk: ⟨constants⟩ (1)

- Jeder Prozess erhält einen eigenen Adressraum
- Threads (eines Prozesses) teilen sich einen Adressraum
- max. 1024 Prozesse
  → max. 1024 Adressräume

⟨constants⟩≡

```
#define MAX_ADDR_SPACES 1024
```

Chunk: ⟨elementary type definitions⟩ (1)

- Wir verwenden häufig Address-Space-IDs (als Index in die Adressraum-Tabelle)
- definieren eigenen Typ `addr_space_id` (`unsigned int`)

⟨elementary type definitions⟩≡

```
typedef unsigned int addr_space_id;
```

Chunk: ⟨more TCB entries⟩ (1)

- Jeder TCB verweist auf einen Adressraum
- Feld zur TCB-Struktur hinzufügen

⟨more TCB entries⟩≡

```
addr_space_id addr_space;  // memory usage
```

Speicher-Layout für Prozesse

Address Range	Usage	Access
0xD4000000 – 0xFFFFFFFF	*unused*	–
0xD0000000 – 0xD3FFFFFF	64 MByte Physical RAM (mapped)	K
0xC0000000 – 0xCFFFFFFF	Kernel Code and Data	K
0xBFFFF000 – 0xBFFFFFFF	Kernel Stack (4 KByte = one page)	K
0xB0000000 – 0xBFFFEFFF	*unused*	–
… – 0xAFFFFFFF	User Mode Stack	U
	Heap (can be grown with `sbrk`)	(U)
0x00000000 – …	Process Code and Data	U

Chunk: ⟨constants⟩ (2)

Speicher-Layout für Prozesse:

- Code / Daten / Heap:
  0 - ...
- User Mode Stack:
  ... - `0xB0000000` (exkl.)
- Kernel Mode Stack:
  ... - `0xC0000000` (exkl.)
- Kernel Code / Daten:
  `0xC0000000` - ...

⟨*constants*⟩+≡

```
#define BINARY_LOAD_ADDRESS 0x0
#define TOP_OF_USER_MODE_STACK 0xb0000000
#define TOP_OF_KERNEL_MODE_STACK 0xc0000000
```

Chunk: ⟨type definitions⟩ (1)

⟨*type definitions*⟩≡

```
typedef struct {
  void         *pd;        // pointer to the page directory
  int          pid;        // process ID (if used by a process; -1 if not)
  short        status;     // are we using this address space?
  unsigned int memstart;   // first address below 0xc000.0000
  unsigned int memend;     // last  address below 0xc000.0000
  unsigned int stacksize;  // size of user mode stack
  unsigned int kstack_pt;  // stack page table (for kernel stack)
  unsigned int refcount;   // how many threads use this address space?
} address_space;
```

Chunk: ⟨constants⟩ (3)

- Zustände eines AS-Eintrags

⟨*constants*⟩+≡

```
#define AS_FREE   0
#define AS_USED   1
#define AS_DELETE 2
```

Chunk: ⟨global variables⟩ (1)

- Adressraum-Tabelle (Array),
- alles auf 0 setzen

⟨*global variables*⟩≡

```
address_space address_spaces[MAX_ADDR_SPACES] = ↩
…{ 0 };
```

Chunk: ⟨function prototypes⟩ (1)

- Funktion für die Suche nach einem freien Adressraum
- einfache Suche, Kriterium:
  `status == AS_FREE`

⟨*function prototypes*⟩≡

```
int get_free_address_space ();
```

Chunk: ⟨function implementations⟩ (1)

⟨*function implementations*⟩≡

```
int get_free_address_space () {
  addr_space_id id = 0;
  while ((address_spaces[id].status != AS_FREE) && (id)) id++;
  if (id==MAX_ADDR_SPACES) id = -1;
  return id;
}
```

Chunk: ⟨enable paging for the kernel⟩ (1)

- Adressraum 0 für Kernel

⟨*enable paging for the kernel*⟩≡
```
  address_spaces[0].status   = AS_USED;
  address_spaces[0].pd       = &kernel_pd;
  address_spaces[0].pid      = -1;         // not ↩
…a process
```

Chunk: ⟨function prototypes⟩ (2)

- Neuen Adressraum erzeugen
- `initial_ram`: User Mode Speicher
- `initial_stack`: User Mode Stack

⟨*function prototypes*⟩+≡
```
  int create_new_address_space (int initial_ram, i↩
…nt initial_stack);
```

Chunk: ⟨macro definitions⟩ (1)

⟨*macro definitions*⟩≡
```
  #define MAKE_MULTIPLE_OF_PAGESIZE(x)  x = ((x+PAGE_SIZE-1)/PAGE_SIZE)*PAGE_SIZE
```

(Einfaches Makro, das Größenangaben so anpasst, dass sie immer ein Vielfaches der Seitengröße sind)

Chunk: ⟨function implementations⟩ (2)

⟨*function implementations*⟩+≡
```
  int create_new_address_space (int initial_ram, int initial_stack) {
    MAKE_MULTIPLE_OF_PAGESIZE (initial_ram);
    MAKE_MULTIPLE_OF_PAGESIZE (initial_stack);
    // reserve address space table entry
    addr_space_id id;
    if ( (id = get_free_address_space()) == -1 )  return -1;    // fail
    address_spaces[id].status    = AS_USED;
    address_spaces[id].memstart  = BINARY_LOAD_ADDRESS;
    address_spaces[id].memend    = BINARY_LOAD_ADDRESS + initial_ram;
    address_spaces[id].stacksize = initial_stack;
    address_spaces[id].refcount  = 1;  // default: used by one process
    ⟨reserve memory for new page directory⟩    // sets new_pd
    address_spaces[id].pd = new_pd;
    ⟨copy master page directory to new directory⟩

    int frameno, pageno;  // used in the following two code chunks
    if (initial_ram > 0)   {  ⟨create initial user mode memory⟩   }
    if (initial_stack > 0) {  ⟨create initial user mode stack⟩      }
    return id;
  };
```

Chunk: ⟨reserve memory for new page directory⟩ (1)

⟨*reserve memory for new page directory*⟩≡
```
  page_directory *new_pd = (void*)request_new_page ();
  if (new_pd == NULL) {  // Error
    printf ("\nERROR: no free page, aborting create_new_address_space\n");
    return -1;
  };
  memset (new_pd, 0, sizeof(page_directory));
```

Chunk: ⟨copy master page directory to new directory⟩ (1)

⟨copy master page directory to new directory⟩≡

```
*new_pd = kernel_pd;
memset ((char*)new_pd, 0, 4);    // clear first entry (kernel pd contains
                                 // old reference to some page table)
```

Chunk: ⟨function prototypes⟩ (3)

- Im nächsten Code Chunk werden wir die Funktion `as_map_page_to_frame()` verwenden
- Implementation später...

⟨function prototypes⟩+≡

```
int as_map_page_to_frame (int as, unsigned int p↩
…ageno, unsigned int frameno);
```

Chunk: ⟨create initial user mode memory⟩ (1)

- User Mode Speicher:
- richtige Anzahl Frames reservieren
- und in die Seitentabelle eintragen

⟨create initial user mode memory⟩≡

```
pageno = 0;
while (initial_ram > 0) {
  if ((frameno = request_new_frame ()) < 0) {
    printf ("\nERROR: no free frame, aborting cr↩
…eate_new_address_space\n");
    return -1;
  };
  as_map_page_to_frame (id, pageno, frameno);
  pageno++;
  initial_ram -= PAGE_SIZE;
};
```

Chunk: ⟨create initial user mode stack⟩ (1)

- User Mode Stack:
- richtige Anzahl Frames reservieren
- und in die Seitentabelle eintragen
- diesmal zählt die Schleife runter (Stack wächst nach unten, feste Anfangsadresse)

⟨create initial user mode stack⟩≡

```
pageno = TOP_OF_USER_MODE_STACK / PAGE_SIZE;
while (initial_stack > 0) {
  if ((frameno = request_new_frame ()) < 0) {
    printf ("\nERROR: no free frame, aborting cr↩
…eate_new_address_space\n");
    return -1;
  };
  pageno--;
  as_map_page_to_frame (id, pageno, frameno);
  initial_stack -= PAGE_SIZE;
}
```

Chunk: ⟨function implementations⟩ (3)

⟨*function implementations*⟩+≡

```
int as_map_page_to_frame (int as, unsigned int pageno, unsigned int frameno) {
  // for address space as, map page #pageno to frame #frameno
  page_table* pt;
  page_directory* pd;

  pd = address_spaces[as].pd;           // use the right address space
  unsigned int pdindex = pageno/1024;   // calculuate pd entry
  unsigned int ptindex = pageno%1024;   // ... and pt entry

  if ( ! pd->ptds[pdindex].present ) {
    // page table is not present
    ⟨create new page table for this address space⟩   // sets pt
  } else {
    // get the page table
    pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
  };
  if (pdindex < 704)   // address below 0xb0000000 -> user access
    UMAP ( &(pt->pds[ptindex]), frameno << 12 );
  else
    KMAP ( &(pt->pds[ptindex]), frameno << 12 );
  return 0;
};
```

Chunk: ⟨create new page table for this address space⟩ (1)

⟨*create new page table for this address space*⟩≡

```
int new_frame_id = request_new_frame ();
unsigned int address = PHYSICAL (new_frame_id << 12);
pt = (page_table *) address;
memset (pt, 0, sizeof(page_table));
UMAPD ( &(pd->ptds[pdindex]), new_frame_id << 12);
```

Chunk: ⟨function prototypes⟩ (4)

- Adressraum zerstören (nach Prozessende)
- relativ komplex: alle reservierten Frames bzw. Pages wieder freigeben
- Problem: Kernel-Stack → den nutzen wir gerade!
- Lösung: Eintragen in *kstack delete list*

⟨*function prototypes*⟩+≡

```
void destroy_address_space (addr_space_id id);
```

Chunk: ⟨function implementations⟩ (4)

⟨*function implementations*⟩+≡

```
void destroy_address_space (addr_space_id id) {
  // called only from syscall_exit(), holding thread_list_lock, interrupts off
  if (--address_spaces[id].refcount > 0) return;
  addr_space_id as = current_as;          // remember current address space
  current_as = id;                        // set current_as: we call release_page()

  ⟨destroy AS: release user mode pages⟩  // all pages used by the process
  ⟨destroy AS: release user mode stack⟩   // all its user mode stack pages
  ⟨destroy AS: release page tables⟩        // the page tables (0..703)

  current_as = as;                        // restore current_as
  address_spaces[id].status = AS_DELETE;  // change AS status

  // remove kernel stack (cannot do this here, this stack is in use right now)
  add_to_kstack_delete_list (id);
  return;
}
```

Chunk: ⟨destroy AS: release user mode pages⟩ (1)

⟨*destroy AS: release user mode pages*⟩≡

```
for ( int i = address_spaces[id].memstart / PAGE_SIZE;
      i < address_spaces[id].memend / PAGE_SIZE;
      i++ ) {
  release_page (i);
};
```

Chunk: ⟨destroy AS: release user mode stack⟩ (1)

⟨*destroy AS: release user mode stack*⟩≡

```
for ( int i = TOP_OF_USER_MODE_STACK / PAGE_SIZE - 1;
      i > (TOP_OF_USER_MODE_STACK-address_spaces[id].stacksize) / PAGE_SIZE - 1;
      i-- ) {
  release_page (i);
};
```

Chunk: ⟨destroy AS: release page tables⟩ (1)

⟨*destroy AS: release page tables*⟩≡

```
page_directory *tmp_pd = address_spaces[id].pd;
for ( int i = 0;  i < 704;  i++ ) {
  if ( tmp_pd->ptds[i].present )
    release_frame ( tmp_pd->ptds[i].frame_addr );
}
```

Chunk: ⟨function prototypes⟩ (5)

- *kstack delete list*: Funktion zum Eintragen
- Bearbeiten der Aufträge: im Scheduler

⟨*function prototypes*⟩+≡

```
  void add_to_kstack_delete_list (addr_space_id id↩
…);
```

Chunk: ⟨constants⟩ (4)

- maximal 1024 Einträge

⟨*constants*⟩+≡

```
  #define KSTACK_DELETE_LIST_SIZE 1024
```

Chunk: ⟨global variables⟩ (2)

- *kstack delete list* ist Array von Adressraum-IDs
- Zugriff darauf mit Lock schützen

⟨*global variables*⟩+≡

```
  addr_space_id kstack_delete_list[KSTACK_DELETE_L↩
…IST_SIZE] = { 0 };
  lock kstack_delete_list_lock;
```

Chunk: ⟨initialize kernel global variables⟩ (1)

- Lock initialisieren
- (mehr zu Locks: später, Thema *Synchronisation*)

⟨*initialize kernel global variables*⟩≡

```
  kstack_delete_list_lock = get_new_lock ("kstack"↩
…);
```

Chunk: ⟨function implementations⟩ (5)

⟨*function implementations*⟩+≡

```
void add_to_kstack_delete_list (addr_space_id id) {
  int i;
  LOCK (kstack_delete_list_lock);
  for (i = 0;  i < KSTACK_DELETE_LIST_SIZE;  i++) {
    // try to enter it here
    if (kstack_delete_list[i] == 0) {
      // found a free entry
      kstack_delete_list[i] = id;
      break;
    }
  }
  UNLOCK (kstack_delete_list_lock);
  if (i == KSTACK_DELETE_LIST_SIZE)
    printf ("ERROR ADDING ADDRESS SPACE TO KSTACK DELETE LIST!\n");
}
```

Chunk: ⟨scheduler: free old kernel stacks⟩ (1)

⟨*scheduler: free old kernel stacks*⟩≡

```
// check all entries in the to-be-freed list
int            i, entry, frameno;
page_directory *tmp_pd;
page_table     *tmp_pt;
LOCK (kstack_delete_list_lock);
for (entry=0; entry) {
  if (kstack_delete_list[entry] != 0 && kstack_delete_list[entry] != current_as) {
    // remove it
    addr_space_id id = kstack_delete_list[entry];
    tmp_pd = address_spaces[id].pd;
    tmp_pt = (page_table *) address_spaces[id].kstack_pt;
    // this is the page table which maps the last 4 MB below 0xC0000000
    for (i=0; i) {
      frameno = tmp_pt->pds[1023-i].frame_addr;
      release_frame (frameno);
    }
    kstack_delete_list[entry] = 0;   // remove entry from kstack delete list

    release_page (((unsigned int)tmp_pt) >> 12);  // free memory for page table
    release_page (((unsigned int)tmp_pd) >> 12);  // ... and page directory
    address_spaces[id].status = AS_FREE;          // mark address space as free
  }
}
UNLOCK (kstack_delete_list_lock);
```

Chunk: ⟨constants⟩ (5)

- Wie groß ist der Kernel Stack?
- 16 KByte (4 Seiten)

⟨*constants*⟩+≡

```
// kernel stack (per process): 1 page = 4 KByte
#define KERNEL_STACK_PAGES 4
#define KERNEL_STACK_SIZE PAGE_SIZE * KERNEL_STA↩
…CK_PAGES
```

Chunk: ⟨function prototypes⟩ (6)

- Aktivieren eines Adressraums:
- Laden der **phys.** Page-Directory-Adresse in Register `CR3`
- Vorsicht: tauscht auch den gerade verwendeten Kernel Stack aus
- darum Funktion nicht aus dem Scheduler heraus aufrufen

⟨function prototypes⟩+≡

```
  inline void activate_address_space (addr_space_i↩
…d id)  __attribute__((always_inline));
```

Chunk: ⟨global variables⟩ (3)

- `current_as`: aktueller Adressraum

⟨global variables⟩+≡

```
  addr_space_id current_as = 0;  // global variabl↩
…e: current address space
  addr_space_id tmp_as;          // temp. address ↩
…space variable, for context switch
```

Chunk: ⟨function implementations⟩ (6)

⟨function implementations⟩+≡

```
  inline void activate_address_space (addr_space_id id) {
    // NOTE: Do not call this from the scheduler; where needed, replicate the code
    unsigned int virt = (unsigned int)address_spaces[id].pd;  // get PD address
    unsigned int phys = mmu (0, virt);            // and find its physical address
    asm volatile ("mov %0, %%cr3" : : "r"(phys)); // write CR3 register
    current_as = id;                              // set current address space
    current_pd = address_spaces[id].pd;           // set current page directory
    return;
  };
```

Chunk: ⟨function prototypes⟩ (8)

Adressübersetzung

- Funktion der MMU nachbilden
- `mmu_p()`: einer Seite ihren Frame zuordnen
- `mmu()`: einer virt. Adresse die phys. Adresse zuordnen
- jeweils bezogen auf einen bestimmten Adressraum
- Aufruf möglich, *ohne* den Adressraum vorher zu wechseln

⟨function prototypes⟩+≡

```
  unsigned int mmu_p (addr_space_id id, unsigned i↩
…nt pageno); // pageno -> frameno
  unsigned int mmu (addr_space_id id, unsigned int↩
… vaddress); // virt. -> phys. addr.
```

Chunk: ⟨function implementations⟩ (8)

⟨*function implementations*⟩+≡

```
unsigned int mmu_p (addr_space_id id, unsigned int pageno) {
  unsigned int pdindex, ptindex;
  page_directory *pd;
  page_table     *pt;
  pdindex = pageno/1024;
  ptindex = pageno%1024;
  pd = address_spaces[id].pd;
  if ( ! pd->ptds[pdindex].present ) {
    return -1;
  } else {
    pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
    if ( pt->pds[ptindex].present ) {
      return pt->pds[ptindex].frame_addr;
    } else {
      return -1;
    };
  }
};
```

Chunk: ⟨function implementations⟩ (9)

- `mmu()` verwendet einfach `mmu_p()`
- vorher Offset ausblenden
- nachher wieder drauf addieren
- (oberste 20 Bits: Seitennummer; untere 12 Bits: Offset)

⟨*function implementations*⟩+≡

```
  unsigned int mmu (addr_space_id id, unsigned int↩
… vaddress) {
    unsigned int tmp = mmu_p (id, (vaddress >> 12)↩
…);
    if (tmp == -1)
      return -1;  // fail
    else
      return (tmp << 12) + (vaddress % PAGE_SIZE);
  }
```

Chunk: ⟨function prototypes⟩ (9)

- Adressraum vergrößern mit `u_sbrk()`
- fügt eine oder mehrere Seiten hinzu (Heap)

⟨*function prototypes*⟩+≡

```
  void *u_sbrk (int incr);
```

Chunk: ⟨function implementations⟩ (10)

⟨*function implementations*⟩+≡

```
  void *u_sbrk (int incr) {
    int pages = incr / PAGE_SIZE;
    int i, frame;
    address_space *aspace = &address_spaces[current_as];

    unsigned int oldbrk = aspace->memend;

    for (i=0; i) {
      frame = request_new_frame ();
      if (frame == -1) { return (void*)(-1); } // error!
      as_map_page_to_frame (current_as, aspace->memend/PAGE_SIZE, frame);
      aspace->memend += PAGE_SIZE;
    }
    return (void*) oldbrk;
  }
```

Chunk: ⟨syscall prototypes⟩ (1)

- Zugehöriger System Call Handler

⟨*syscall prototypes*⟩≡
```
void syscall_sbrk (context_t *r);
```

Chunk: ⟨syscall functions⟩ (1)

⟨*syscall functions*⟩≡
```
void syscall_sbrk (context_t *r) {
  // ebx: increment
  r->eax = (unsigned int)u_sbrk (r->ebx);
  return;
}
```

Chunk: ⟨initialize syscalls⟩ (1)

- Syscall eintragen

⟨*initialize syscalls*⟩≡
```
install_syscall_handler (__NR_brk, syscall_sbrk)↩
…;
```

Thread Control Block

- `TCB`: zentrale Datenstruktur für Threads / Prozesse
- bei ULIX: Prozess-ID (oder Thread-ID) identisch mit Position in TCB-Liste
- enthält Feld vom Typ `context_t` für Context Switch
- Referenz auf Adressraum haben wir schon gesehen
- Thread-Liste: Array von TCBs

Chunk: ⟨type definitions⟩ (2)

⟨*type definitions*⟩+≡
```
typedef struct {
  thread_id     tid;         // thread id
  thread_id     ppid;        // parent process
  int           state;       // state of the process
  context_t     regs;        // context
  unsigned int  esp0;        // kernel stack pointer
  unsigned int  eip;         // program counter
  unsigned int  ebp;         // base pointer
  ⟨more TCB entries⟩
} TCB;
```

Chunk: ⟨constants⟩ (6)

- maximal 1024 Threads

⟨*constants*⟩+≡
```
#define MAX_THREADS 1024
```

Chunk: ⟨global variables⟩ (4)

- Thread-Tabelle

⟨*global variables*⟩+≡
```
TCB thread_table[MAX_THREADS];
```

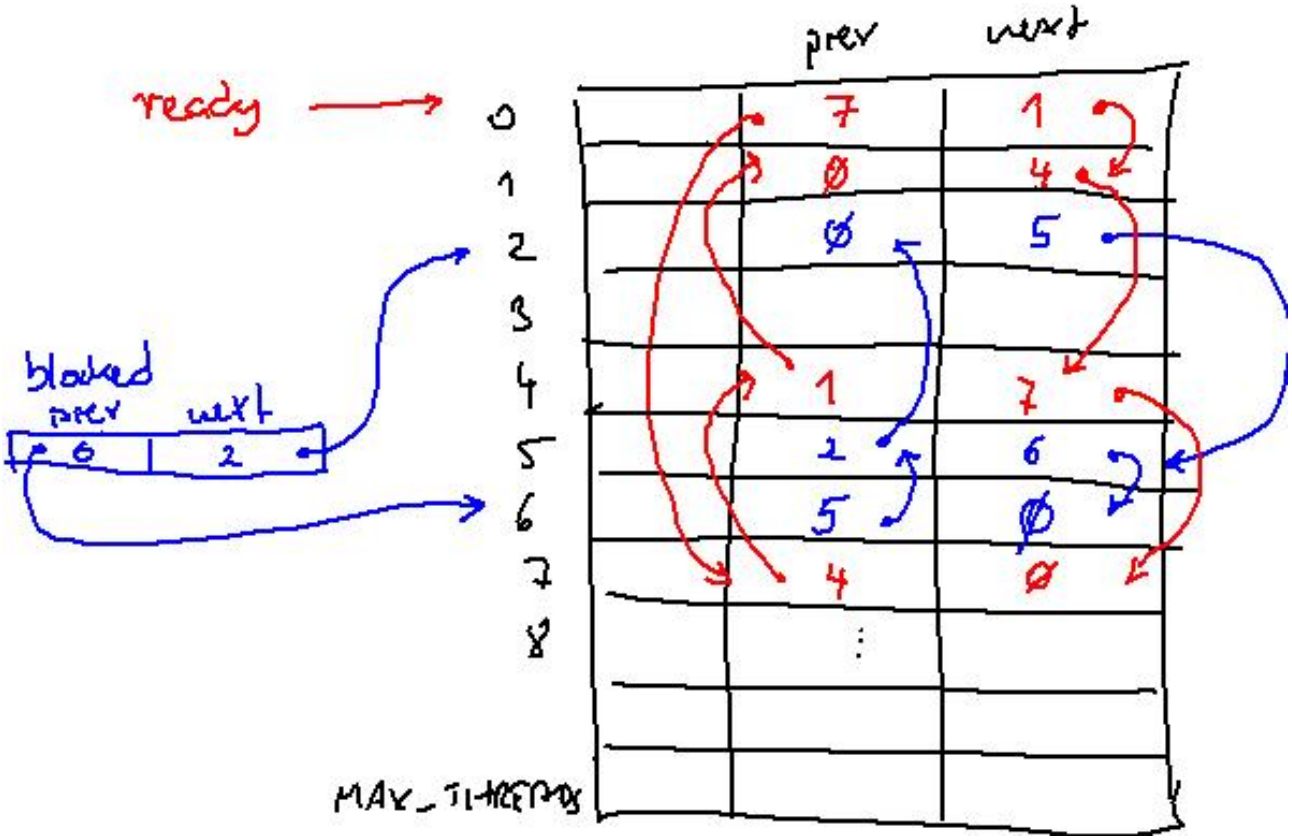
Chunk: ⟨more TCB entries⟩ (2)

- Wir brauchen Listen von Threads
- z. B. Liste der bereiten Threads,
- mehrere Blockiert-Listen
- Verwaltung über Zeiger `next`, `prev`; direkt im TCB
- ready queue: `thread_table[0].next` definiert Anfang (0 ist keine gültige Thread-ID!)

⟨more TCB entries⟩+≡

```
thread_id next; // id of the ``next'' thread
thread_id prev; // id of the ``previous'' thread
```

Listen-Verwaltung



(Bild: Felix Freiling)

Chunk: ⟨declaration of blocked queue⟩ (1)

- Blockiert-Liste:
- Struktur, die via `next` und `prev` auf TCBs zeigt

⟨declaration of blocked queue⟩≡

```
typedef struct {
  thread_id next; // id of the ``next'' thread
  thread_id prev; // id of the ``previous'' thre↩
…ad
} blocked_queue;
```

Chunk: ⟨function implementations⟩ (11)

⟨function implementations⟩+≡

```
void initialize_blocked_queue (blocked_queue* q) {
  q->prev = 0;
  q->next = 0;
}
```

Chunk: ⟨function prototypes⟩ (10)

- Bereit-Warteschlange:
- Thread hinzufügen (setzt auch Status auf `TSTATE_READY`)
- Thread entfernen

⟨*function prototypes*⟩+≡
```
void add_to_ready_queue (thread_id t);
void remove_from_ready_queue (thread_id t);
```

Chunk: ⟨function implementations⟩ (12)

⟨*function implementations*⟩+≡
```
void add_to_ready_queue (thread_id t) {
  thread_id last = thread_table[0].prev;
  thread_table[0].prev = t;
  thread_table[t].next = 0;
  thread_table[t].prev = last;
  thread_table[last].next = t;
  thread_table[t].state = TSTATE_READY;  // set its state to ready
}
```

Chunk: ⟨function implementations⟩ (13)

⟨*function implementations*⟩+≡
```
void remove_from_ready_queue (thread_id t) {
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  thread_table[prev_thread].next = next_thread;
  thread_table[next_thread].prev = prev_thread;
}
```

Chunk: ⟨initialize kernel global variables⟩ (2)

- Initialisierung: Bereit-Warteschlange ist leer

⟨*initialize kernel global variables*⟩+≡
```
thread_table[0].prev = 0;
thread_table[0].next = 0;
```

Chunk: ⟨function prototypes⟩ (11)

⟨*function prototypes*⟩+≡
```
void add_to_blocked_queue        (thread_id t, blocked_queue* bq);
void remove_from_blocked_queue   (thread_id t, blocked_queue* bq);
thread_id front_of_blocked_queue (blocked_queue bq);
```

Chunk: ⟨function implementations⟩ (14)

⟨*function implementations*⟩+≡
```
thread_id front_of_blocked_queue (blocked_queue bq) {
  return bq.next;
}
```

Chunk: ⟨function implementations⟩ (15)

⟨*function implementations*⟩+≡

```
void add_to_blocked_queue (thread_id t, blocked_queue* bq) {
  thread_id last = bq->prev;
  bq->prev = t;
  thread_table[t].next = 0; // [[t]] is ``last'' thread
  thread_table[t].prev = last;
  if (last == 0) {
    bq->next = t;
  } else {
    thread_table[last].next = t;
  }
}
```

Chunk: ⟨function implementations⟩ (16)

⟨*function implementations*⟩+≡

```
void remove_from_blocked_queue (thread_id t, blocked_queue* bq) {
  thread_id prev_thread = thread_table[t].prev;
  thread_id next_thread = thread_table[t].next;
  if (prev_thread == 0) {
    bq->next = next_thread;
  } else {
    thread_table[prev_thread].next = next_thread;
  }
  if (next_thread == 0) {
    bq->prev = prev_thread;
  } else {
    thread_table[next_thread].prev = prev_thread;
  }
}
```

Chunk: ⟨more TCB entries⟩ (3)

- Verwaltung der TCBs
- Feld used markiert, ob TCB frei ist

⟨*more TCB entries*⟩+≡

```
boolean used;
```

Chunk: ⟨global variables⟩ (5)

- Wir wollen Thread-IDs fortlaufend vergeben
- d.h.: auch wenn ein Thread beendet wird, vergeben wir dessen Nummer (zunächst) nicht neu
- später (wenn alle Nummern belegt waren) schon
- Suche beginnt immer bei `next_pid`

⟨*global variables*⟩+≡

```
int next_pid = 1;
```

Chunk: ⟨find free TCB entry⟩ (1)

⟨*find free TCB entry*⟩≡

```
boolean tcbfound = false;
int tcbid;
for ( tcbid=next_pid; ((tcbid) && (!tcbfound)); tcbid++ ) {
  if (thread_table[tcbid].used == false) {
    tcbfound = true;
    break;
  }
};
```

Chunk: ⟨find free TCB entry⟩ (2)

⟨*find free TCB entry*⟩+≡

```
if (!tcbfound) {                                  // continue searching at 1
  for ( tcbid=1; ((tcbid) && (!tcbfound)); tcbid++ ) {
    if (thread_table[tcbid].used == false) {
      tcbfound = true;
      break;
    }
  };
};

if (tcbfound) next_pid = tcbid+1;                 // update next_pid:
// either tcbfound == false or tcbid == index of first free TCB
```

Chunk: ⟨function prototypes⟩ (12)

- Funktion verknüpft TCB und Adressraum

⟨*function prototypes*⟩+≡

```
int register_new_tcb (addr_space_id as_id);
```

Chunk: ⟨function implementations⟩ (18)

⟨*function implementations*⟩+≡

```
int register_new_tcb (addr_space_id as_id) {
  // called by ulix_fork() which aquires LOCK (thread_list_lock)
  ⟨find free TCB entry⟩
  if (!tcbfound) {
    return -1; // no free TCB!
  };
  thread_table[tcbid].used       = true;   // mark as used
  thread_table[tcbid].addr_space = as_id;  // enter address space ID
  return tcbid;
}
```

Start des ersten Prozesses

Schritte:

1. TCB und Adressraum reservieren
2. Speicher reservieren (User Mode Code/Data/Stack + Kernel Mode Stack)
3. Programm von Platte laden
4. TCB aktualisieren
5. Datenstruktur TSS (→ später) erzeugen
6. Sprung in den User Mode / Aktivierung des Prozess'

Chunk: ⟨global variables⟩ (6)

- Variable für aktuellen Thread,
- vgl. `current_as` für aktuellen Adressraum

⟨*global variables*⟩+≡

```
volatile int current_task;
```

Chunk: ⟨constants⟩ (7)

- init-Prozess (u.a. Shell): max. 32 K
- könnte man auch variabel gestalten (erhöht Komplexität)

⟨*constants*⟩+≡

```
#define PROGSIZE 32768
```

Chunk: ⟨global variables⟩ (7)

- Zugriff auf TCBs mit Lock schützen

⟨*global variables*⟩+≡

```
  lock thread_list_lock = NULL;     // initialize ↩
…this when the first process starts
```

Chunk: ⟨function implementations⟩ (19)

⟨*function implementations*⟩+≡

```
  void start_program_from_disk (char *progname) {
    if (thread_list_lock == NULL)               // initialize lock for thread list
      thread_list_lock = get_new_lock ("thread list");

    ⟨start program from disk: prepare address space and TCB entry⟩
    ⟨start program from disk: load binary⟩
    ⟨start program from disk: create kernel stack⟩

    current_task = tid;                         // make this the current task
    add_to_ready_queue (tid);                   // add process to ready queue
    ENABLE_SCHEDULER;
    cpu_usermode (BINARY_LOAD_ADDRESS,
                  TOP_OF_USER_MODE_STACK);      // jump to user mode
  };
```

Chunk: ⟨start program from disk: prepare address space and TCB entry⟩ (1)

⟨*start program from disk: prepare address space and TCB entry*⟩≡

```
  addr_space_id as;
  thread_id tid;
  as = create_new_address_space(64*1024, 4096);   // 64 KB + 4 KB stack
  tid = register_new_tcb (as);                    // get a fresh TCB
  thread_table[tid].tid = tid;
  thread_table[tid].ppid = 0;                     // parent: 0 (none)
  thread_table[tid].new = false;                  // not freshly created via fork()
  thread_table[tid].terminal = 0;                 // default terminal: 0
  memcpy (thread_table[tid].cwd,     "/",   2);   // set current directory
  memcpy (thread_table[tid].cmdline, "new", 4);   // set temporary command line
  activate_address_space (as);                    // activate the new address space
```

Chunk: ⟨start program from disk: load binary⟩ (1)

⟨*start program from disk: load binary*⟩≡

```
  // read binary
  int mfd = mx_open (DEV_HDA, progname, O_RDONLY);
  mx_read (DEV_HDA, mfd, (char*)BINARY_LOAD_ADDRESS, PROGSIZE);
                                                  // load to virtual address 0
  mx_close (DEV_HDA, mfd);
```

Chunk: ⟨start program from disk: create kernel stack⟩ (1)

⟨start program from disk: create kernel stack⟩≡

```
unsigned int framenos[KERNEL_STACK_PAGES];   // frame numbers of kernel stack pages
int i; for (i=0; i)
  framenos[i] = request_new_frame();
page_table* stackpgtable = (page_table*)request_new_page();
memset (stackpgtable, 0, sizeof(page_table));
KMAPD ( ¤t_pd->ptds[767], mmu (0, (unsigned int)stackpgtable) );
for (i=0; i)
  as_map_page_to_frame (current_as, 0xbffff - i, framenos[i]);
char* kstack = (char*) (TOP_OF_KERNEL_MODE_STACK-KERNEL_STACK_SIZE);
unsigned int adr = (uint)kstack;                    // one page for kernel stack
tss_entry.esp0 = adr+KERNEL_STACK_SIZE;

thread_table[tid].esp0 = (uint)kstack + KERNEL_STACK_SIZE;
thread_table[tid].ebp  = (uint)kstack + KERNEL_STACK_SIZE;
```

Chunk: ⟨install GDTs for User Mode⟩ (1)

- bisher: Code- und Datensegmente `0x08`, `0x10` (Kernel)
- jetzt: Code- und Datensegmente `0x18`, `0x20` (User Mode)
- `0xFA` = 1**11**1 1010 (bin)
- `0xF2` = 1**11**1 0010 (bin)
- DPL: 3 = **11** (bin)
  Executable? 1 = yes

⟨install GDTs for User Mode⟩≡

```
fill_gdt_entry (3, 0, 0xFFFFFFFF, 0xFA, 0b1100);
fill_gdt_entry (4, 0, 0xFFFFFFFF, 0xF2, 0b1100);
```

Chunk: ⟨type definitions⟩ (3)

⟨type definitions⟩+≡

```
typedef struct {
  unsigned int prev_tss    : 32;  // unused: previous TSS
  unsigned int esp0, ss0   : 32;  // ESP and SS to load when we switch to ring 0
  long long u1, u2         : 64;  // unused: esp1, ss1, esp2, ss2 for rings 1 and 2
  unsigned int cr3         : 32;  // unused: page directory
  unsigned int eip, eflags : 32;
  unsigned int eax, ecx, edx, ebx, esp, ebp, esi, edi, es, cs, ss, ds, fs, gs : 32;
                                  // unused (dynamic, filled by CPU)
  long long u3             : 64;  // unused: ldt, trap, iomap
} __attribute__((packed)) tss_entry_struct;
```

⟨global variables⟩+≡

```
tss_entry_struct tss_entry;
```

Chunk: ⟨install GDTs for User Mode⟩ (2)

- GDT erhält auch einen Eintrag für die TSS
- `write_tss`-Argumente:
  - GDT-Nummer,
  - SS0, ESP0

⟨install GDTs for User Mode⟩+≡

```
write_tss (5, 0x10, 0xc0000000);
```

Chunk: ⟨function implementations⟩ (20)

⟨*function implementations*⟩+≡

```
void write_tss (int num, unsigned short ss0, unsigned int esp0) {
   unsigned int base = (unsigned int) &tss_entry;
   unsigned int limit = sizeof (tss_entry) - 1;
   fill_gdt_entry (num, base, limit, 0xE9, 0b0000);   // enter it in GDT

   memset (&tss_entry, 0, sizeof(tss_entry));        // fill with zeros

   tss_entry.ss0  = ss0;  // Set the kernel stack segment.
   tss_entry.esp0 = esp0; // Set the kernel stack pointer.
}
```

- `0xE9` = 1**11**0 1001 (bin)
- DPL: 3 = **11** (bin)
  0: Das ist kein normaler GDT-Eintrag (sondern TSS)

Chunk: ⟨start.asm⟩ (1)

- `tss_flush()`: in Assembler-Datei
- muss Assembler-Befehl `ltr` (load *task register*) ausführen

⟨*start.asm*⟩≡

```
[section .text]
                global tss_flush

tss_flush:      mov ax, 0x28 | 0x03
                ltr ax                  ; load the tas↩
…k register
                ret
```

Chunk: ⟨function prototypes⟩ (14)

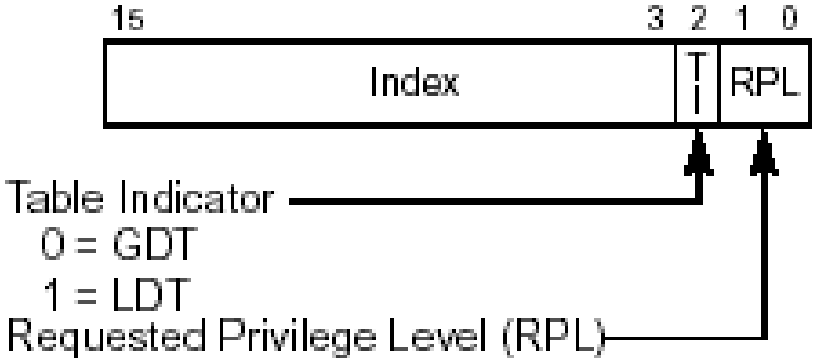
- Umschalten auf User Mode (Ring 3): mit Assembler-Funktion `cpu_usermode()`
- Trick: Stack so vorbereiten, dass `iret`-Aufruf zum Wechsel in Ring 3 führt
- auf Stack liegen dann u. a. die Segmentregister
- und die enthalten auch den einzustellenden Protection Level

⟨*function prototypes*⟩+≡

```
extern void cpu_usermode (unsigned int address, ↩
…unsigned int stack);   // assembler
```

Segment-Selektoren

- `iret` liest Werte auf dem Stack und füllt damit u. a. die Segment-Register
- Diese passend einstellen → schaltet User Mode ein



(Bild: http://www.cs.cmu.edu/~410/doc/segments/segments.html)

Chunk: ⟨start.asm⟩ (2)

⟨*start.asm*⟩+≡

```
; code from http://f.osdev.org/viewtopic.php?t=23890&p=194213 (Jezze)
; modified and comments added
              global cpu_usermode
cpu_usermode: cli                   ; disable interrupts
              mov  ebp, esp         ; remember current stack address
              mov  ax,  0x20 | 0x03 ; code selector 0x20 | RPL3: 0x03
                                    ; RPL = requested protection level
              mov  ds,  ax
              mov  es,  ax
              mov  fs,  ax
              mov  gs,  ax
              mov  eax, esp
              push 0x20 | 0x03      ; code selector 0x20 | RPL3: 0x03
              mov  eax, [ebp + 8]   ; stack address is 2nd argument
              push eax              ; stack pointer
              pushf                 ; EFLAGS
              pop  eax              ; trick: reenable interrupts when doing iret
              or   eax, 0x200
              push eax
              push 0x18 | 0x03      ; code selector 0x18 | RPL3: 0x03
              mov  eax, [ebp + 4]   ; return address (1st argument) for iret
              push eax
              iret
```