So we wrote a microkernel once …

… and it was ChaOS

Adrian Herrmann

Unleash the ChaOS

- Operating systems course at Freie Universität Berlin
- Developed over 14 weeks, 7 milestones
  - Biweekly assignments
- Team effort with two fellow students
  - 100% collaborative VS Code Live Share
  - Full credit to everyone
- Free choice of design, code style, coding practices, …

Target platform

- taskit Portux MiniPC
- AT91RM9200 SoC
- ARM920T CPU (ARMv4T)

- 16 MiB Flash, 64 MiB RAM
- Ethernet, serial, USB host/target
- LEDs and 4*16 text display

- U-Boot bootloader in Flash

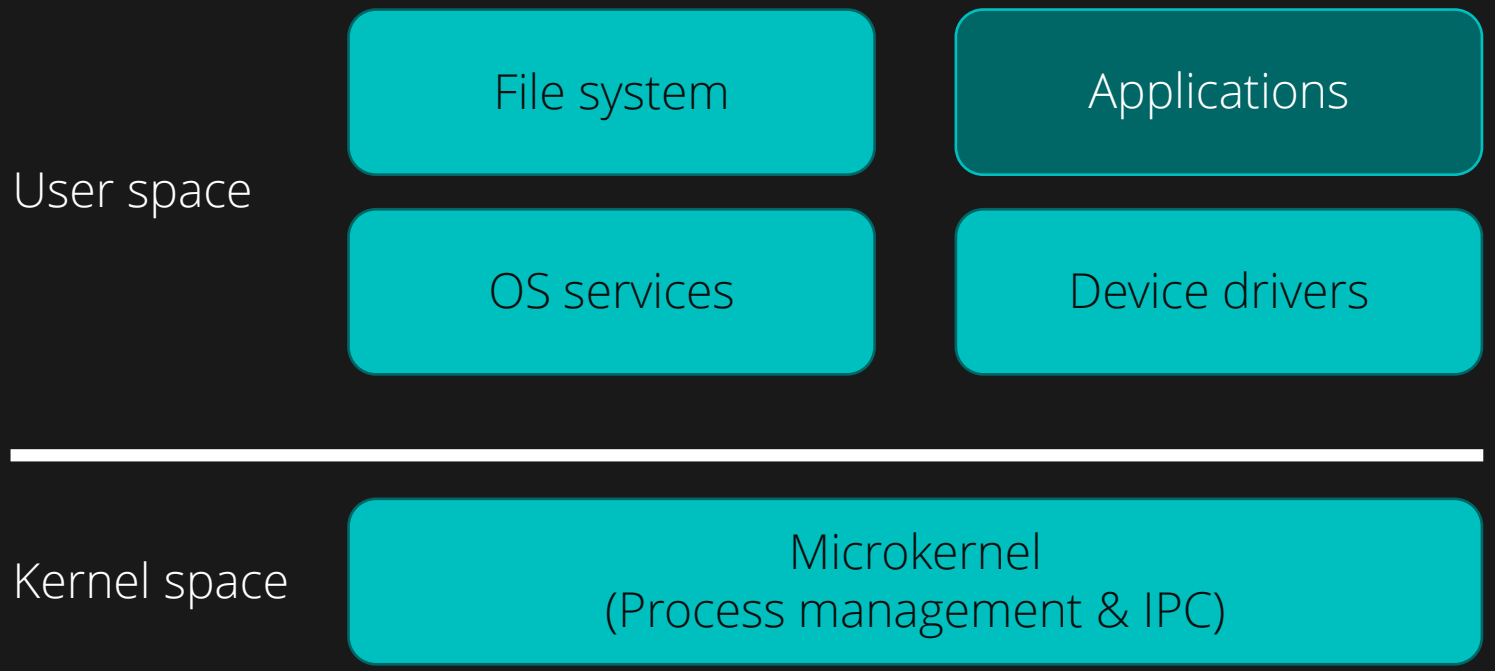
- Emulated with a patched QEMU

What is an operating system?

- Management & operation of physical resources
- Abstraction of physical with logical resources
- Process management (scheduling, switching, ...)
- Inter-process communication
- Interface for programs

What is a microkernel?

- Only the most necessary parts of the OS are in the kernel & in kernel space
- The rest is in user space – even some OS services!
  - E.g., part of scheduling



What can ChaOS do?

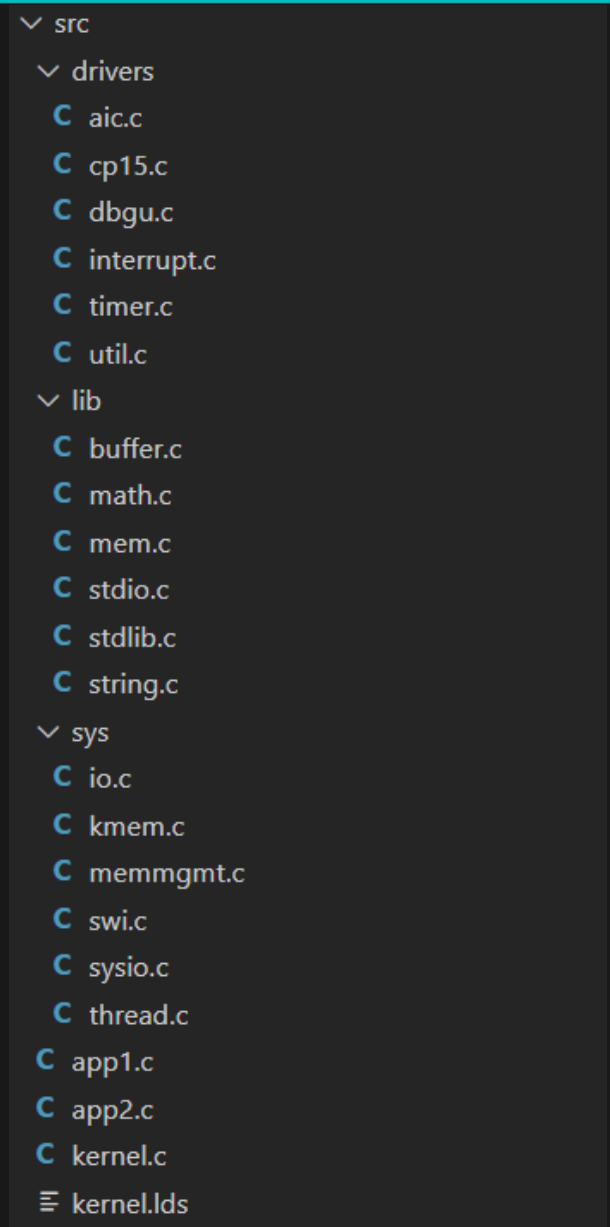
- Serial driver (read/write, interrupts)
- Processor modes, stacks, dynamic kernel memory management
- Interrupt handlers (swi, abort, fiq, irq)
- System timer & scheduling
- Processes/threads, preemptive context switching
- Memory protection, address spaces (MMU)
- User/kernel interface (syscalls, utility library)

Limitations

- No filesystem
  - Everything happens in RAM!
- No dynamic loading of code
  - Everything is statically linked into kernel binary

Project structure

- drivers: device drivers and functions that interact with hardware directly
  - E.g., via memory-mapped I/O
- lib: application libraries, utility functions – hardware-independent!
- sys: OS libraries that are for internal use by the kernel and drivers



Kernel entry function

- System initialization
  - Stacks, serial port, Interrupt Vector Table, interrupts, MMU, thread management, timer
  - CP15: Coprocessor for control of cache, TLB, MMU
- Thread creation: Program code is statically linked!
- Kernel now only runs for interrupts!

```
27  __attribute__((naked, section(".init")))
28  void _start() {
29      init_stacks();
30
31      // Init the kernel memory management
32      // kmem_init(KMEM_START, KMEM_SIZE);
33
34      io_dbgu_init();
35
36      interrupt_enable();
37
38      dbgu_enable();
39      printf_isr("DBGU has been enabled.\n");
40
41      dbgu_rxrdy_interrupt_enable();
42      printf_isr("DBGU RXRDY Interrupt has been enabled.\n");
43
44      printf_isr("Create Interrupt Vector Table and initialize system.\n");
45      init_ivt();
46
47      printf_isr("Initializing Advanced Interrupt Controller.\n");
48      aic_enable_system_peripherals();
49
50      printf_isr("Initializing allocation table.\n");
51      memmgmt_init_allocation_table();
52
53      printf_isr("Initializing thread management.\n");
54      thread_init_management();
55
56      printf_isr("Initializing CP15 domains.\n");
57      cp15_init_domains();
58
59      printf_isr("Welcome to ChaOS.\n");
60
61      struct thread_tcb* thread = thread_create(&main, 0, 0, 0);
62      if (thread) {
63          thread_activate(thread->id);
64      }
65
66      // timer_init_periodical(32768);
67      timer_init_real_time(32);
68      timer_init_periodical(32);
69      // Nothing should be executed after this line
70
71      while(1);
72  }
```

The idle thread

- If the kernel only runs for interrupts, what to do if there are no threads? → idle thread

- Threads can yield processing time before scheduler kicks in
  - Idle thread does “nothing”, for as little time as possible

```
36  /**
37   * The main function to be executed by the idle thread.
38   */
39  __attribute__((section(".lib")))
40  void thread_idle_text(void) {
41      // The idle thread makes a repeated SWI_THREAD_YIELD call to
42      // ensure it only runs when there is no other thread to yield to.
43      while(1) {
44          asm volatile(
45              "swi 0x20"
46              ::
47          );
48      }
49  }
```

Thread switching

- Part of normal IRQ service routine
- Detect if IRQ came from system timer
  - Therefore, preemptive multitasking
- If so, call thread switching routine

```
133  /**
134   * Interrupt Request (IRQ)
135   */
136  __attribute__ ((interrupt ("IRQ")))
137  void isr_interrupt_request(void) {
138      char c;
139      struct thread_tcb* thread;
140
141      // Interrupt from the Period Interval Timer
142      if (timer_read_PIT_status()) {
143          thread_unblock_for_timer();
144          thread_switch();
145          return;
146      }
147  ...
```

Thread switching

- Every thread gets a time slot (round-robin)
- Save thread context in Thread Control Block
- Restore context from next thread
- Thread selection separate from switching – sometimes we just want selection without the entire rest!
  - E.g., when an swi needs to block the current thread

```
292  /**
293   * Switches the running thread.
294   *
295   * Use only in the IRQ Interrupt Service Routine!
296   */
297  __attribute__((always_inline))
298  inline void thread_switch(void) {
299
300      // Check that there is a thread currently running
301      if (thread_tcb_list[thread_sched_cur_idx].status == THREAD_STATUS_RUNNING) {
302          // Do not switch if the thread has not worked through its time slot yet
303          if (thread_switch_counter++ < THREAD_ROUND_ROBIN_TIME_SLOT) {
304              return;
305          }
306          thread_switch_counter = 0;
307
308          // Save the current thread's context and set its status
309          thread_save_context(&thread_tcb_list[thread_sched_cur_idx]);
310          thread_tcb_list[thread_sched_cur_idx].status = THREAD_STATUS_READY;
311      }
312
313      thread_select();
314
315      thread_restore_context(&thread_tcb_list[thread_sched_cur_idx]);
316      thread_tcb_list[thread_sched_cur_idx].status = THREAD_STATUS_RUNNING;
317
318  }
```

Thread context saving

```
116  __attribute__((always_inline))
117  inline uint32_t* thread_get_fp(void) {
118      uint32_t* fp;
119      asm volatile (
120          "mov %[fp], fp \n\t"
121          : [fp] "=r" (fp)
122      );
123      return fp;
124  }
```

- Some registers automatically pushed on the thread's stack by CPU when IRQ
- Others plainly available or shadowed (r13-14)

- Saving frame pointer essential
  - Lots of inline code for thread switching!

```
126  /**
127   * Saves the context of the last running thread into its TCB.
128   * Use only in the IRQ Interrupt Service Routine!
129   *
130   * @param tcb               A pointer to the thread's TCB
131   */
132  __attribute__((always_inline))
133  inline void thread_save_context(struct thread_tcb* tcb) {
134
135      uint32_t* ptr;
136      uint8_t i;
137      uint32_t* b;
138      uint32_t s;
139
140      ptr = thread_get_fp() - 6;
141      for (i = 0; i < 4; i++) { // r0-r3
142          tcb->r[i] = ptr[i];
143      }
144
145      tcb->r[11] = ptr[4]; // fp r11
146      tcb->r[12] = ptr[5]; // ip r12
147      tcb->r[15] = ptr[6]; // pc r15
148
149      b = &tcb->r[4];
150      asm volatile ( // r4-r10
151          "stm %[rs], {r4-r10} \n\t"
152          : [rs] "=r" (b)
153      );
154
155      b = &tcb->r[13];
156      asm volatile ( // r13-r14
157          "stm %[rs], {r13-r14}^ \n\t"
158          : [rs] "=r" (b)
159      );
160
161      s = tcb->r[THREAD_REG_CPSR];
162      asm volatile ( // cpsr
163          "mrs %[rs], SPSR \n\t"
164          : [rs] "=r" (s)
165      );
166
167  }
```

Software interrupt handler

- Run desired swi function withing software interrupt handler
- Thread selection outside scheduling!
  - E.g., swi_getc() blocks thread and calls to select next one
  - Still, can't forget saving the context!

```c
/**
 * Software Interrupt (SWI)
 */
__attribute__ ((interrupt ("SWI")))
void isr_software_interrupt(void) {

    void* iptr = read_link_register() - 4;
    uint32_t inst = *(uint32_t*)iptr & 0xFF;
    uint8_t i = 0;

    struct thread_tcb* tcb = thread_get_current();
    thread_save_context(tcb);

    while (swi_types[i++]) {
        if (inst == swi_types[i-1]) {
            ((func)swi_functions[i-1])(tcb);

            tcb = thread_get_current();
            tcb->status = THREAD_STATUS_RUNNING;
            thread_restore_context(tcb);

            return;
        }
    }

    printf_isr("Unknown software interrupt 0x%x detected at address 0x%p.\n", inst, iptr);

}
```

```c
void swi_getc(struct thread_tcb* tcb) {
    thread_block_for_char(tcb);
    thread_select();
}
```

Assembling the thing

- Quite some asm code – no way around it!
  - But part of the fun and challenge :)

```c
/**
 * Enables the IRQ signal.
 */
void interrupt_enable_irq(void) {

    asm volatile (
        "mrs r3, CPSR \n\t"
        "and r3, r3, #0xFFFFFF7F \n\t"
        "msr CPSR, r3 \n\t"
    );

}

/**
 * Enables the FIQ signal.
 */
void interrupt_enable_fiq(void) {

    asm volatile (
        "mrs r3, CPSR \n\t"
        "and r3, r3, #0xFFFFFFBF \n\t"
        "msr CPSR, r3 \n\t"
    );

}

/**
 * Disables the IRQ signal.
 */
void interrupt_disable_irq(void) {

    asm volatile (
        "mrs r3, CPSR \n\t"
        "orr r3, r3, #0x80 \n\t"
        "msr CPSR, r3 \n\t"
    );

}

/**
 * Disables the FIQ signal.
 */
void interrupt_disable_fiq(void) {

    asm volatile (
        "mrs r3, CPSR \n\t"
        "orr r3, r3, #0x40 \n\t"
        "msr CPSR, r3 \n\t"
    );

}
```

```c
/**
 * Writes a given number of bytes into the standard output.
 *
 * @param source    Pointer to the buffer to write bytes from
 * @param size      The number of bytes to write
 *
 * @return          The number of bytes written
 */
__attribute__((section(".lib")))
size_t write_string(char* source, size_t size) {
    size_t len;
    asm volatile(
        "mov r7, %[source] \n"
        "mov r8, %[size] \n"
        "swi 0x10 \n"
        "mov %[len], r7"
        : [len] "=r" (len)
        : [source] "r" (source), [size] "r" (size)
        : "r7", "r8"
    );
    return len;
}
```

```c
/**
 * Writes the address of the translation table base to the MMU.
 *
 * @param ptr       The address of the TTB
 */
void cp15_write_translation_table_base(uint32_t* ptr) {

    ptr = (uint32_t*) ((uint32_t)ptr & 0xFFFFC000);
    asm volatile (
        "mov r7, %[ptr] \n"
        "mcr p15, 0, r7, c2, c0, 0 \n"
        :
        : [ptr] "r" (ptr)
        : "r7"
    );

}
```

Documentation

```c
/**
 * Initializes the Domain Access Control Register.
 */
void cp15_init_domains(void) {

    /*
     * The CP 15 Register 3, or Domain Access Control Register, defines the domain's access
     * permission.
     * MMU accesses are controlled through the use of 16 domains.
     * Each field of register 3 is associated with one domain.
     *
     * The 2-bit field value allows domain access as described in the table below.
     *
     * Value    Access      Description
     * -----------------------------------------------------------
     * 00       No access   Any access generates a domain fault
     * 01       Client      Accesses are checked against the access permission
     *                      bits in the section or page descriptor
     * 10       Reserved    Reserved. Currently behaves like the no access mode
     * 11       Manager     Accesses are not checked against the access permission
     *                      bits so a permission fault cannot be generated
     */

    // Currently we are not interested in a case where we wouldn't want to check accesses
    // against the access permission, so we define one single domain where this is the case.
    asm volatile (
        "mov r1, #1 \n"
        "mcr p15, 0, r1, c3, c0, 0 \n"
        : : : "r1"
    );

}
```

```c
#define DBGUB       0xFFFFF200  // Base address

#define DBGU_CR     0x0000      // Control Register                (WRITE ONLY)
#define DBGU_MR     0x0004      // Mode Register                   (READ/WRITE)    (RESET VALUE 0x00)
#define DBGU_IER    0x0008      // Interrupt Enable Register       (WRITE ONLY)
#define DBGU_IDR    0x000C      // Interrupt Disable Register      (WRITE ONLY)
#define DBGU_IMR    0x0010      // Interrupt Mask Register         (READ ONLY)     (RESET VALUE 0x00)
#define DBGU_SR     0x0014      // Status Register                 (READ ONLY)
#define DBGU_RHR    0x0018      // Receive Holding Register        (READ ONLY)     (RESET VALUE 0x00)
                                //
                                // This holds only one value inside the first eight bits, the rest is always empty.
                                //
                                // RXCHR: Received Character
                                // Last received character if RXRDY is set.
                                //
                                // We just need to address one byte beginning in DBGU_RHR and don't need any defines
                                // for this register.
#define DBGU_THR    0x001C      // Transmit Holding Register       (WRITE ONLY)
                                //
                                // This holds only one value inside the first eight bits, the rest is always empty.
                                //
                                // TXCHR: Character to be Transmitted
                                // Next character to be transmitted after the current character if TXRDY is not set.
                                //
                                // We just need to address one byte beginning in DBGU_THR and don't need any defines
                                // for this register.
#define DBGU_BRGR   0x0020      // Baud Rate Generator Register    (READ/WRITE)    (RESET VALUE 0x00)
                                //
                                // This holds only one value inside the first 16 bits, the rest is always empty.
                                //
                                // CD: Clock Divisor
                                //
                                // CD           Baud Rate Clock
                                // ----------------------------
                                // 0            Disabled
                                // 1            MCK
                                // 2 to 65535   MCK / (CD * 16)
                                //
                                // We just need to address two bytes beginning in DBGU_BRGR and don't need any defines
                                // for this register.

// Reserved memory space from 0x0024 - 0x003C

#define DBGU_CIDR   0x0040      // Chip ID Register                (READ ONLY)
#define DBGU_EXID   0x0044      // Chip ID Extension Register      (READ ONLY)
                                // Adding further specifications for these two registers would add a tremendous
                                // overhead to this file for something that is most definitely not going to be
                                // needed.
```

- 😵😵‍💫
- Good old manual work – but important 😉