μC/OS-II

Mike Holenderski

Department of Mathematics and Computer Science
System Architecture and Networking
Eindhoven University of Technology

07 April 2014
Outline

• Introduction to operating systems
• Introduction to μC/OS-II
• Tasks
• Scheduling
• Interrupts
• Exercises
There are standards (or specifications) which operating systems should satisfy, in order to provide the desired portability, e.g. POSIX.
Operating system

- Provide applications with a hardware abstraction layer
  - Generic interfaces (APIs) hiding hardware details
  - Convenient abstractions, addressing common problems
    - File system, unix pipes, tasks, ...
  - Multiplex application tasks on the shared resources
    - Processor, bus, network, memory, timers, ...

- Monolithic vs. microkernel based operating system
  - Microkernel:
    - Kernel implements only basic services (task and memory management, and task communication)
    - Higher level services are implemented on top
    - Increased maintainability, security and stability

The microkernel design allows for easier management of code due to its division into user space services. This also allows for increased security and stability resulting from the reduced amount of code running in kernel mode. For example, if a networking service crashed due to buffer overflow, only the networking service's memory would be corrupted, leaving the rest of the system still functional.
Real-time operating system

• Manage tasks and communication between tasks
  - Interrupt handling
  - Task scheduling
  - Context switching between tasks
  - Task synchronization and communication:
    • Semaphores, mutexes, timers, ...

• Known and high performance
  - low and **bounded latencies and jitter** for API calls and ISRs

• Small memory foot print (for embedded use)
  - Configurable (no cost for unused functionality)

Our focus:
Tasks (with suspension)
Timer management (activating tasks)
Example: OSEK/VDX

- Joint project in German/French automotive industry
- Motivation:
  - High, recurring expenses in the development of control software (i.e. non-application)
  - Incompatibility of control units made by different manufactures due to different interfaces and protocols
- Goal: “create an industry standard for an open-ended architecture for distributed control units in vehicles”
  - Support for portability and reusability, through:
    - Specification of abstract interfaces (i.e. independent of applications and hardware)
    - Configurable and efficient architecture

http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=4&Itemid=4

OSEK (1993): BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW, University of Karlsruhe
OSEK/VDX (1994): PSA, Renault
Example: OSEK/VDX

- Several OSEK specifications:
  - Operating System
    • Real-time execution of ECU software and base for the other OSEK/VDX modules
  - Communication
    • Data exchange within and between ECUs
  - Network Management
    • Configuration and monitoring
Example: OSEK OS

- Task management
  - Basic and Extended tasks
  - Activation, suspension and termination of tasks
  - Task switching
  - Note: tasks, semaphores, ... must be statically allocated!

- Synchronization
  - Priority Ceiling

- Interrupt management

- Alarms (i.e. Timers)

- Error treatment

Extended tasks are like basic tasks, but can react to external events.
OSERK OS: task states

Extended task

- **running**
  - terminate
  - preempt
  - start

- **suspended**
  - preempt
  - activate

- **ready**
  - release
  - preempt

Basic task

- **running**
  - terminate

- **suspended**
  - preempt
  - activate

- **ready**
  - preempt

Figure 4-2: States and status transitions for extended tasks

Figure 4-1: States and status transitions for basic tasks
Example: OSEK OS

- Task management
  - Basic and Extended tasks
  - Activation, suspension and termination of tasks
  - Task switching
- Synchronization
  - Priority Ceiling
- Interrupt management
- Alarms (i.e. Timers)
- Error treatment
What is μC/OS-II?

- Real-time operating system
- Used in medical, military, aerospace, automotive, consumer electronics, ...
- Commercial, but “open source”
Properties of μC/OS-II

• **Fixed-priority preemptive** multitasking
  - Up to 64 or 256 tasks (configurable)

• **Small** and **deterministic** overheads
  - Short and predictable interrupt path

• **Scalable**
  - Many services: semaphores, mutexes, flags, mailboxes, ...
  - Enable services with conditional compilation directives

• **Nested** interrupts
  - Up to 255 levels deep
μC/OS-II architecture

“μC/OS–II Port” contains platform specific code, such as context switching and interrupt initialization. Porting μC/OS–II to a new platform requires to change only the “μC/OS–II Port”.

This presentation is about “Application Software” and “μC/OS–II”
Application structure

```c
void main(void) {
    OSInit();
    OSTaskCreate(Task1,
                  (void*)100,
                  (void*)&Task1Stack[TASK_STACK_SIZE-1],
                  4);
    OSTaskCreate(Task2, ...);
    RandomSem = OSSemCreate(1);
    OSStart();
}
```

main():
- initializes the system with OSInit(), e.g. internal data structures, interrupt handlers, ...
- creates the tasks with OSTaskCreate() (akin to RegisterTask() in Johan’s kernel)
- starts multitasking with OSStart()

NOTE: uC/OS-II allows **dynamic creation** of tasks, semaphores. **In this example** they are created statically (similar to OSEK).
Each task has its own state (to support blocking).
Task example

OS_STK Task1Stack[TASK1_STACK_SIZE];

void Task1 (void* data) {
    INT8U err;
    INT16U delay = data;
    INT16U x;
    for (x = 0;; x++) {
        OSSemPend(RandomSem, 0, &err);
        if (x % 3 == 0) ToggleLeds(GREEN);
        OSTimeDly(delay);
    }
}

void main(void) {
    ...
    OSTaskCreate(Task1, 50,
        &Task1Stack[TASK1_STACK_SIZE-1], 5);
    RandomSem = OSSemCreate(0);
    ...
}

Tasks are implemented as repetitive loops.

NOTE: the stack can grow up or down, depending on the underlying architecture, i.e. how the push() and pop() methods work.
Task: one shot vs. thread

• Both are dispatched by the scheduler within an interrupt handler

• One shot
  - **Cannot** block or self suspend
  - Executed within the context of the interrupt handler

• Thread
  - **Can** block or self suspend
  - Preserves its context
    • Has its **own stack**

One shot tasks are executed from scratch upon every interrupt handler invocation (just like the tasks in Johan’s kernel).
Task: one shot vs. thread

- Both are dispatched by the scheduler within an interrupt handler

- One shot
  - Cannot block or self suspend
  - Executed within the context of the interrupt handler

✓ Thread
  - Can block or self suspend
  - Preserves its context
  - Has its own stack

Johan’s kernel supports one shot tasks.

uC/OS-II implements threads.
The state of a task is described by two kinds of variables: Task Control Block stores task variables which are common in all tasks. Task context contains the rest, e.g. local variables used during task execution, special registers.
Task state storage

void Task1(void) {
    a = ReadRotation();
    if (a < A)
        ActivateABS();
}

void Task2(void) {
    p = ReadPressure();
    if (p > P)
        InflateAirbag();
}
Preemptive kernels are used when responsiveness is important (the highest ready task is always given control of the CPU, as long as data consistency is preserved).

Unique priorities avoid round robin and simplify resource sharing.

OSSched() is called from synchronization primitives. OSIntExit() is called at the end of an interrupt handler, in particular the tick interrupt handler.
Scheduler

• How does the scheduler select the highest priority task?
  - Maintain a ready queue keeping track which tasks are ready and which are not
  - Maintain the queue when tasks become ready or not ready
  - When scheduler is invoked, consult the queue to select the highest priority task
Ready queue implementation

- A list ordered by priority
  - Task becomes ready: insert task into the list
  - Task becomes not ready: remove task from the list
  - Select highest priority task: take the task from the head

- A bitmap, with one bit per task
  - Bit is 1 means task is ready, bit is 0 means task is not ready
  - Task becomes ready: set the corresponding bit to 1
  - Task becomes not ready: set the corresponding bit to 0
  - Select highest priority task: select the “right most” set bit
The following piece of code is used to place a task in the ready list:

```c
OSRdyGrp            |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];
```

where `prio` is the task's priority.

As you can see from Figure 3-3, the lowest 3 bits of the task's priority are used to determine the bit position in `OSRdyTbl[]`, while the next three significant bits are used to determine the index into `OSRdyTbl[]`.

Note that `OSMapTbl[]` (see `OS_CORE.C`) is a table in ROM, used to equate an index from 0 to 7 to a bit mask as shown in the table below:

<table>
<thead>
<tr>
<th>Index</th>
<th>Bit mask (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
</tr>
</tbody>
</table>

[Labrosse, 2002]
Scheduler

- The ready state of all tasks is managed with global variables
  - OSRdyTbl[] and OSRdyGrp

- Making a task ready
  OSRdyGrp |= OSMapTbl[prio >> 3];
  OSRdyTbl[prio >> 3] |= OSMapTbl[prio & 0x07];

- Checking if a task is ready
  y = OSUnMapTbl[OSRdyGrp];
  x = OSUnMapTbl[OSRdyTbl[y]];  
Prio = (y << 3) + x;

- Both can be done in constant time!
  - At the memory cost of lookup tables OSMapTbl and OSUnMapTbl

OSMapTbl[] and OSUnMapTbl[] are arrays in ROM mapping between numbers in the range 0..7 and the corresponding bit mask, containing a 1 at the position indicated by the number.

The thing to take away from this slide is that the ready table can be managed in constant time.
Interrupts

• Interrupt handler dispatched by an incoming interrupt

• Has higher priority than any task (interfere with tasks)
  - Needs to have a short and predictable execution time
  - Tasks can disable interrupts
    • `OS_ENTER_CRITICAL()` and `OS_EXIT_CRITICAL()`
    • Can lead to increased interrupt latency or missed interrupts
    • Keep interrupts disabled for as little time as possible!
    • Use only for short critical sections

• Represents high priority events (need to be handled)
  - Sometimes sufficient to only disable the scheduler
    • Task maintains control of the CPU, but incoming interrupts are handled

There is a tradeoff: interrupts should be handled with little latency, but can lead to high interference of other tasks.
Interrupt Service Routine

- General structure of an ISR in μC/OS-II:

```c
void OSTickISR(void) {
    Save processor registers;
    Call OSIntEnter();
    Call OSTimeTick();
    Call OSIntExit();
    Restore processor registers;
    Execute a return from interrupt instruction;
}
```

- ISR executes within the context of the currently running task
  - It uses the stack space of the current task
**ASK:** how would you measure interrupt latency?

interrupt latency = longest critical section

Response time defined slightly different that in Johan’s slides (S05.14) and Reinder’s: the response is the time until the **beginning** of the User ISR, rather than its end. (Also their task response time in the book is until the beginning of the task, rather than its completion).

Scenario A is selected when the same task is resumed after the interrupt.
Scenario B is selected when a higher priority task is ready to execute. Includes a context switch to the higher priority task.
Interrupt timing

- Interrupt latency
  - Max time interrupts are disabled
  - Time to start executing first instruction of the ISR

- Interrupt response
  - Interrupt latency
  - Time to save CPU context (registers PC, PWD)
  - Time to enter the ISR (OSIntEnter())

- Interrupt recovery
  - Time to exit the ISR (OSIntExit())
  - Time to restore CPU context

void SomeISR(void) {
    Save processor registers;
    Call OSIntEnter();
    Call SomeISRBody();
    Call OSIntExit();
    Restore processor registers;
    Return from interrupt;
}

Relate to Slide 29, 14
Time to Start executing the first instruction of ISR includes waiting for last instruction to complete

OSIntExit() includes time to determine if higher priority task is ready and switching to it.
Timer interrupt

- Timer interrupts are especially important
  - Task delays in control applications, network packet timeouts, ...
  - Keep track of time by incrementing the global OSTime variable
Timer interrupt

- System clock is connected to the MCU via a pin
  - Clock operates independently of the MCU
  - Clock sets the pin high periodically with a specified frequency
  - Setting a pin high triggers an interrupt on the MCU
  - Interrupt is handled by an ISR, which sets the pin low
Timer interrupt
(interrupts enabled)
Timer interrupt
(interrupts disabled)
Timer interrupts

• Timer interrupts are especially important
  - Task delays in control applications, network packet timeouts, ...

• Timed evens in uC/OS-II
  - No periodic tasks!
  - Single primitive:
    • Delay a task for number of ticks: OSTimeDly()
      - Relative to the current time
Here is a logical view on the timer interrupts: each task wants to be triggered by its own timer.

The different sizes represent the work that needs to be done by the task.
Most platforms provide a single hardware timer, so we need to multiplex the logical (or software) timers onto the single timer.
The Multiplexer box represents the overhead for multiplexing several software timers on a single hardware timer.

Note that the multiplexer delays the time when a task can start responding to the expiration of its timer.
Timer interrupts

![Diagram showing timer interrupts and tasks](image-url)
Timer interrupts

- Timer interrupts are especially important
  - Task delays in control applications, network packet timeouts, ...

- Timed evens in uC/OS-II
  - No periodic tasks!
  - Single primitive:
    - Delay a task for number of ticks: OSTimeDly()
      - Relative to the current time

- Handled by the tick handler OSTimeTick()
  - Called within the timer interrupt handler
  - Loops through all tasks, decrementing OSTimeDly and checking if it is 0
OSTimeTick()

void OSTimeTick (void) {
    OS_TCB *task;
    
    for all tasks {
        OS_ENTER_CRITICAL();
        if (task->OSTCBDly > 0) {
            if (--task->OSTCBDly == 0) {
                OSRdyGrp |= task->OSTCBBitY;
                OSRdyTbl[task->OSTCBY] |= task->OSTCBBitX;
            }
        }
        OS_EXIT_CRITICAL();
    }
    OS_ENTER_CRITICAL();
    OSTime++;
    OS_EXIT_CRITICAL();
}

Each task is equipped with a OSTCBDly counter: if it is larger than 0 then it represents the number of ticks when the task should become ready.

Most of the work done by OSTimeTick() basically consist of decrementing the OSTCBDly field for each OS_TCB (if it’s nonzero). When the OSTCBDly field of a task’s OS_TCB is decremented to zero, the task is made ready to run.

Note that this is pseudo code with the key parts. The OSTimeTick() function in uC/OS–II also deals with suspended tasks.
OSTimeTick()

```c
void OSTimeTick (void) {
    OS_TCB *task;

    for all tasks {
        OS_ENTER_CRITICAL();
        if (task->OSTCBDly > 0) {
            if (--task->OSTCBDly == 0) {
                OSRdyGrp |= task->OSTCBBitY;
                OSRdyTbl[task->OSTCBY] |= task->OSTCBBitX;
            }
        }
        OS_EXIT_CRITICAL();
    }
    OS_ENTER_CRITICAL();
    OSTime++;
    OS_EXIT_CRITICAL();
}
```

Looping through all the tasks is costly.

Within SAN we have developed a really neat timed event management component, called RELTEQ, and integrated it with uC/OS-II. It avoids looping through all the tasks upon every tick.
Worst-Case Response Time
analysis

• Smallest $x$ satisfying:

$$x = B_i + WC_i + \sum_{j<i} \left[ \frac{x + AJ_j}{WT_j} \right] WC_j$$

• WCRT of the timer interrupt handler:
  - $B_0$ : interrupt latency
  - $WC_0$ : (interrupt response - latency) + OSTimeTick() +
    interrupt recovery
  - $AJ_0$ : clock jitter
  - $WT_0$ : timer interrupt frequency

Note: this analysis holds for deadlines smaller or equal to
periods (i.e. task’s response time needs to be smaller than
its period).

Clock jitter: due to hardware inaccuracies and the divider
(1024 works).
Worst-Case Response Time analysis

• Smallest $x$ satisfying:

$$x = B_i + WC_i + \sum_{j<i} \left\lfloor \frac{x + AJ_j}{WT_j} \right\rfloor WC_j$$

• WCRT of the highest priority task:
  - $B_i$ : longest time a **lower priority** task disabled interrupts
  - $WC_i$ : worst-case path through Task()
  - $AJ_i$ : $AJ_0 + WR_0 - BR_0 = AJ_0 + B_0 + WC_0 - BC_0$
  - $WT_i$ : task period

Need to consider other resources as well: accessing memory, network, ... . In our case we do not allow those.

Activation jitter: assuming no nested interrupts. Clock is never interrupted.
Exercises

• Measure
  - Tick interrupt latency (suggest how to measure)
  - Tick interrupt overhead
    • OSTimeTick() (best and worst case)

• Compare μC/OS-II with Johan’s kernel
  - Measure the above for Johan’s kernel

• Instantiate the WCRT analysis for Johan’s kernel and task set from Exercise C1
Exercises

- Implement periodic tasks on top of uC/OS-II
  
  Given the following task structure:

```c
void Task1 (void* data) {
    INT16U period = data;
    INT16U k = 0;
    INT32U offset = OSTimeGet();
    for (;;) {
        ...
        OS_ENTER_CRITICAL();
        delay = ++k*period - (OSTimeGet() - offset);
        OS_EXIT_CRITICAL();
        OSTimeDly(delay);
    }
}
```

What are the key issues here? Propose an improvement. Hint: use semaphores, and either timers in os_tmr.c or the timer hook OSTimeTickHook() to handle the arrival of periodic events.
Exercise

- Optional (for the very ambitious): implement one-shot tasks in uC/OS-II
References


• “μC/OS-II Reference Manual”
  - Chapter 16 from the book
  - Available online

• μC/OS-II source code
  - Download from http://micrium.com