b
UNIVERSITÄT
BERN

Internet of Things

III. Operating Systems

Prof. Dr. Torsten Braun, Institut für Informatik

Bern, 08.03.2021

Internet of Things: Operating Systems

Table of Contents

1. Introduction

1. Platform Requirements

- Low power consumption
- Concurrency-intensive operation
- Reactivity,
  at least soft-real-time capabilities
- Robustness and self-configurability
- Flexibility: programmability
  and reconfiguration
- Compatibility
- Security and privacy
- Memory limitations
  → low memory footprint
- Network interoperability
- System interoperability

1. Introduction

2. Node Level Platform Options

Node-centric operating system providing hardware and network abstractions of a sensor node to programmers

- Traditional operating systems:
  file management, memory allocation, task scheduling, device drivers, networking etc.
- For sensor nodes: simplified versions of traditional operating systems
- Example: MANTIS OS, RIOT

Node-level programming tools

- Language platform providing library of components to programmers
- Example: TinyOS, Contiki

1. Introduction

3.1 Concurrency

True Concurrency vs. Pseudo-Concurrency

– Low-cost networked embedded systems usually have only 1 CPU core.

– Concurrent activities are mapped to pseudo-concurrent sequential processes.

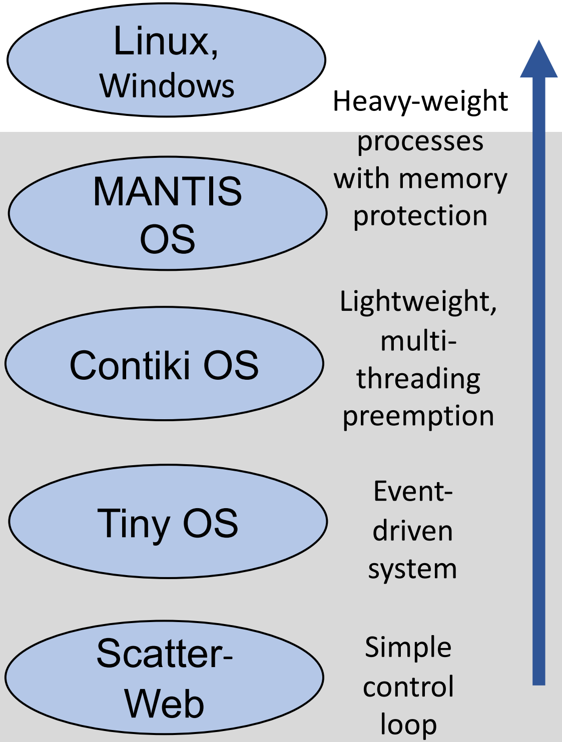
– True concurrency requires true parallel processing units (CPU cores).

Concurrency

– Ability to provide concurrency “costs” CPU and memory.

– First sensor node OSs had almost no concurrency capabilities.

– OSs based on simple control loops proved to be inflexible and died, e.g., ScatterWeb.

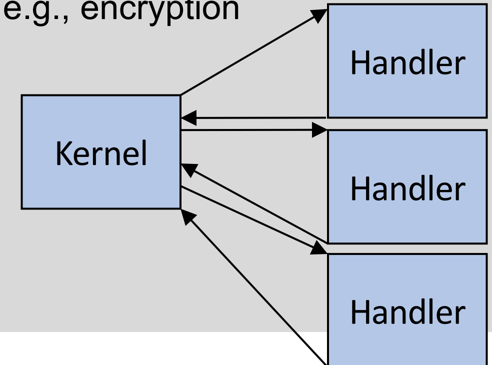


1. Introduction

3.2 Concurrency

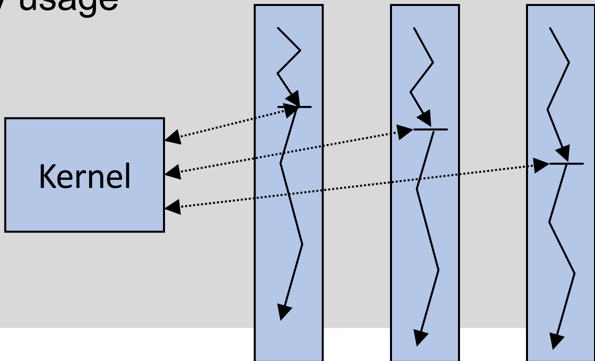
Event-driven

- Processes do not run without events.
- Kernel invokes event handler when event occurs.
- Only one event can run at a time!
- Event handler runs to completion (explicit return).
  - may be a problem for long computations, e.g., encryption



Multi-Threading

- Blocked threads are waiting for events.
- Kernel unblocks threads when event occurs.
- Thread runs until next blocking statement.
- Each thread requires own stack.
- Larger memory usage

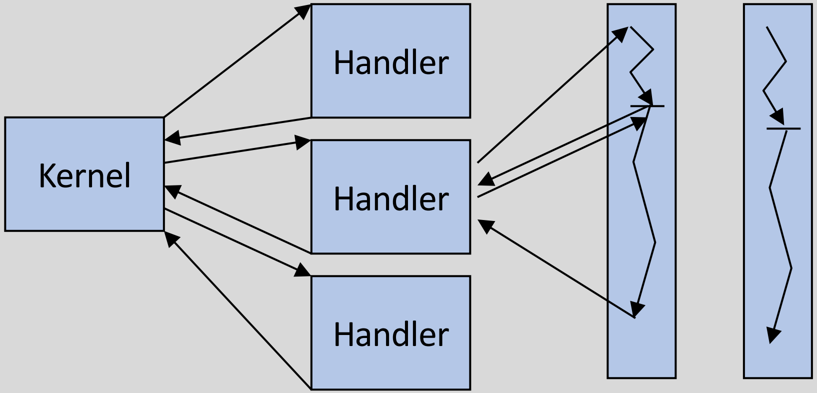


1. Introduction

3.3 Concurrency

Mix: Threads on top of event-driven kernel

- Kernel is event-based, invokes event handler when an event occurs.
- Multi-threading implemented as library
- Threads only used when needed



1. Introduction

4. Event- and Thread-Driven Execution

Event-driven Execution

- \+ (Pseudo) Concurrency with low resources
- \+ complements the way networking protocols work
- \+ Inexpensive scheduling technique
- \+ Highly portable
- \- Event-loop is in control.
- \- Program needs to be chopped to subprograms.
- \- Bounded buffer producer-consumer problem
- \- High learning curve

Thread-driven Execution

- \+ eliminates bounded buffer problem
- \+ Programmer in control of program
- \+ Automatic scheduling
- \+ Real-time performance
- \+ Low learning curve
- \+ simulates parallel execution
- \- Complex shared memory
- \- Expensive context switches
- \- High memory footprint
- \- Not portable due to stack manipulation
- \- performs better on multiprocessors

2. TinyOS

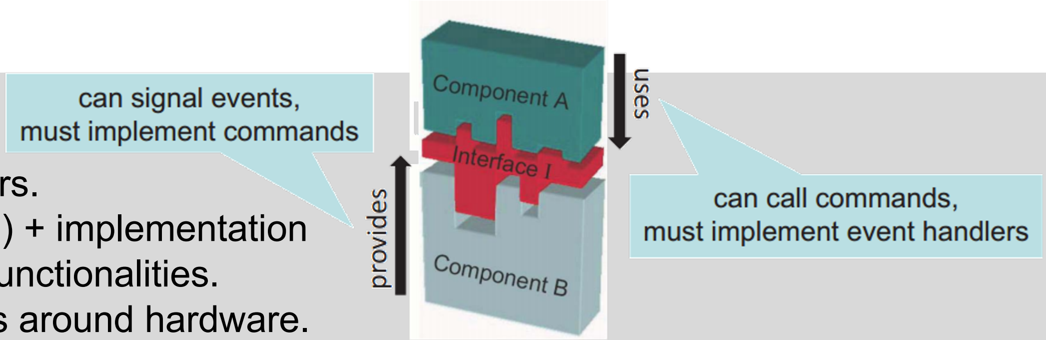
1. Overview

– 1st OS for sensor network applications on resource-constrained HW platforms.
– classical event-based OS
– Simplifications
  – No file system
  – Static memory allocation
  – Implementation of a simple task model
  – Thread support by application level thread library using standard synchronization mechanisms, e.g., semaphores, condition variables
  – Minimal device and network abstractions
– Language-based application development approach
  – Only necessary parts of operating system are compiled with application.
  – or: Each application is built into the operating system.
– Concurrency
  – Concurrency issues are left to the programmer by large extent.
  – Event model allows to handle concurrency in a light-weight fashion → to avoid processor idle times.
  – TinyOS avoids multi-tasking overhead.

UNIVERSITÄT BERN

2. TinyOS

2. Component-based Architecture



- Component-based approach
  - Components are organized into layers.
  - Component = Interface (bidirectional) + implementation
  - Components encapsulate software functionalities.
  - Some components are thin wrappers around hardware.
  - Components are typically implemented as reentrant state machines.
  - All variables are inside components (no dynamic memory allocation).
  - TinyOS provides system software components as set of libraries.
- TinyOS application = scheduler + graph of components
  - Applications are typically developed using a special language (nesC) and wire components together

2. TinyOS

3. nesC

- C extension
- Static language
  - no dynamic memory allocations
  - call graph known at compile time
- Direct support of event-based TinyOS design

- nesC applications are built out of components with well-defined interfaces.
  - Component provides and uses interfaces, which are the only point of access to a component.
  - Bidirectional interfaces: commands and events
- Types of components
  - *Modules* provide application code, implementing one or more interfaces
  - *Configurations* are used to wire other components together
    by connecting component interfaces.

2. TinyOS

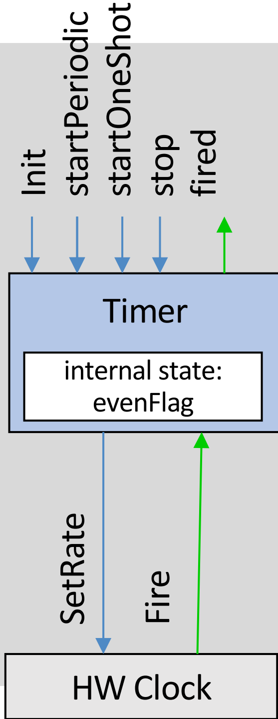
4.1 Example: Timer Component

- Timer can set rate of the clock (wrapper around hardware clock).
- Hardware clock generates periodic interrupts and toggles internal state of timer.
- Timer component fires dependent on internal state.

```
interface Timer<precision_tag> {

   event void fired();

   command void startPeriodic(…);
   command void startOneShot(…);
   command void stop();
   …
}
```



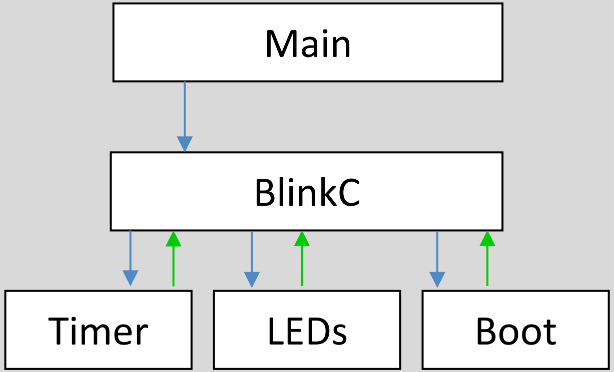
2. TinyOS

4.2 Example: Blink Application

Task: Blink the red LED every 1 second

Implementation: On boot, start a 1 second timer. On timer fire (countdown at 0): toggle state of red LED

```
module BlinkC {
  uses interface Timer<TMilli>
                as BlinkTimer;
  uses interface Leds;
  uses interface Boot;
}
implementation{
  event void Boot.booted() {
    call BlinkTimer.startPeriodic(1000);
  }
  event void BlinkTimer.fired() {
    call Leds.led0Toggle();
  }
}
```



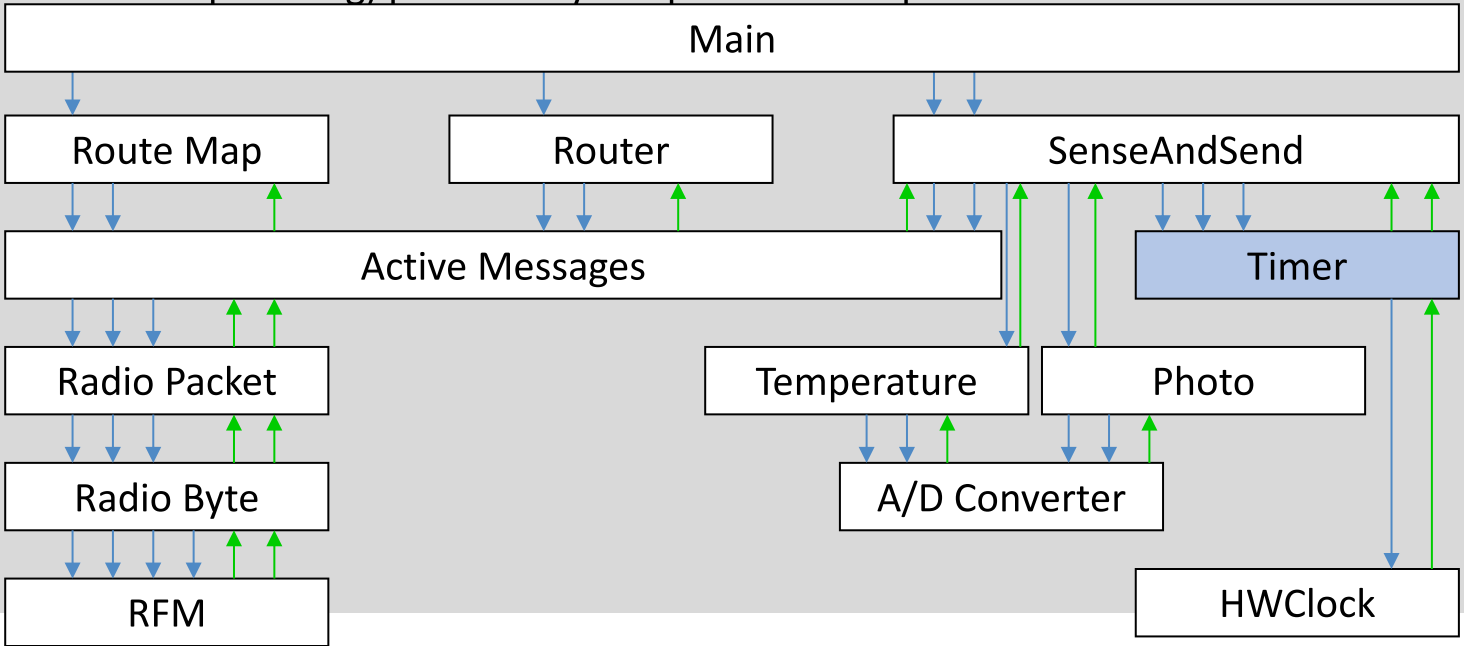
Function calls: Events, Commands

2. TinyOS

4.3 Example: FieldMonitor Application

Nodes send (via multi-hop routing) periodically temperature and photo sensor data to a base station.

Function calls:
Events, commands



UNIVERSITÄT BERN

2. TinyOS

5.1 Events and Tasks

A program executed in TinyOS has two contexts

1. Events
   - Interrupt sources: clock, digital inputs, radio chip, etc.
   - Execution of interrupt handler = event context
   - Processing of events runs to completion, but preempts tasks and can be preempted by other events (last in first out, LIFO).
   - Programmers are required to chop code (in particular in event contexts) into smaller execution pieces to avoid long blocking of tasks
2. Tasks
   - are created by a component and posted to a task scheduler.
   - are deferred computations.
   - always run to completion without preempting other tasks or being preempted by other tasks.

Default TinyOS scheduler

- maintains a task queue and invokes tasks in posting order (FIFO).
- puts node into sleep state if no tasks are available in the queue.

2. TinyOS

5.2 Split-Phase Operation

Separation of method call initiation and return of call (similar to asynchronous method / function calls)

- Client call returns immediately without performing body of call.
- Server executes operation later.
- Server signals completion by calling an event handler in client component.

Example: Packet transmission (in Active Messages component) would block system for a long time.

- `send()` returns immediately.
- Message transmission: conversion of packet to bytes and bits, driving radio circuit
- Caller is notified by `sendDone()` about completion.

UNIVERSITÄT BERN

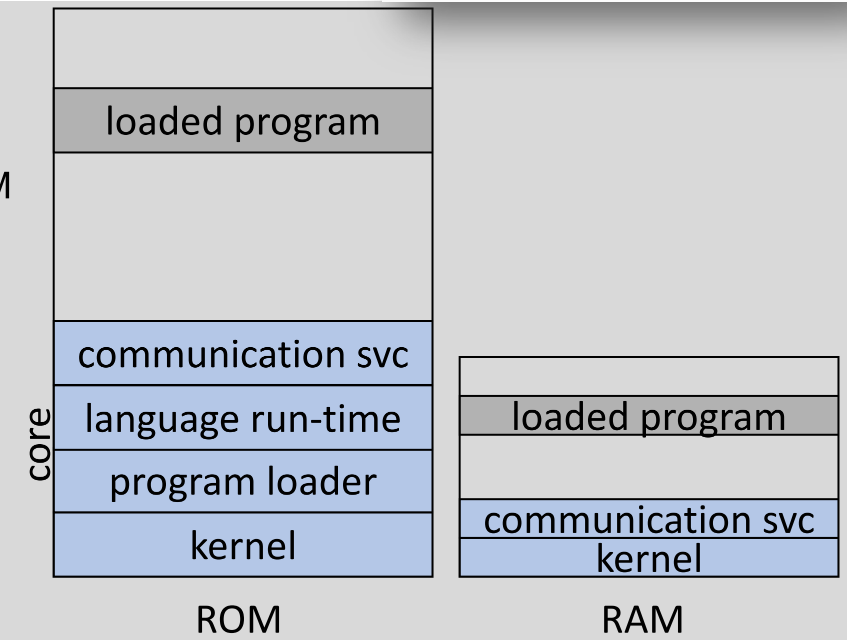
3. Contiki OS

1. Overview

Contiki

The Open Source OS for the Internet of Things

- Event-driven kernel but with preemptive multitasking support
- Code size
  - Typical memory footprint: 40 KB ROM, 2 KB RAM
  - TCP/IP: < 5 KB code, < 2 KB RAM
- Ported to dozens of platforms incl. computers, video game consoles, sensors, pico-satellites
- System parts
  - Core
    - compiled into single binary image
  - Loaded programs
    - loaded by program loader
    - Program obtained via communication stack or EPROM



UNIVERSITÄT BERN

3. Contiki OS

2. Event-Driven Multitasking Kernel

- Contiki System = Kernel + Libraries + set of processes
- Kernel dispatches events to running processes.
- Process is defined by
  - an event handler and
  - an optional polling handler function to poll hardware state.
- Processes are implemented as Protothreads.
- Process state is kept in its own memory.
  Kernel keeps a pointer to the state.
- Processes share the same address space and do not run in different protection domains.
- Single shared stack for process execution
- Inter-process communication:
  Processes post events to each other.

3. Contiki OS

2.1 Lightweight Event Scheduler

dispatches **events** (posted by kernel or polling mechanism) to processes (FIFO).

- Event types
  - Asynchronous:
    - Events are queued by kernel and dispatched to target process later (deferred procedure call).
  - Synchronous:
    - Immediate scheduling of target process and return of control after target has finished event processing (inter-process procedure call)

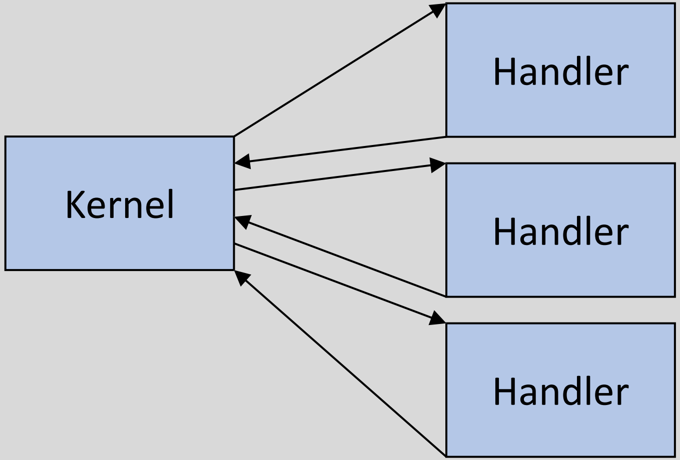
calls processes' **polling handlers** periodically.

- Polls = high priority events scheduled between asynchronous events to check hardware device status
- For a scheduled poll, all processes implementing a poll handler are called according to their priority.

3. Contiki OS

2.2 Preemption and Interrupts

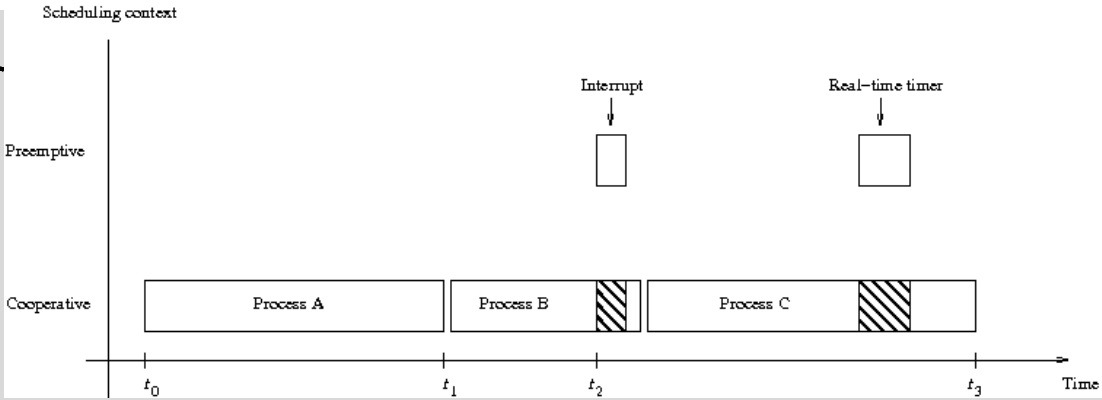
- Event handlers can not preempt each other.
- Event handlers run to completion.
- Preemption of events by interrupts only.
- Interrupts are never disabled.
- Interrupt handlers are not allowed to post events to avoid race conditions but can request immediate polling.



3. Contiki OS

2.3 Execution Contexts

- Code in Contiki runs in either of two execution contexts.
  - cooperative (Contiki processes)
  - preemptive (Interrupts, real-time timers)
- Preemptive code temporarily stops cooperative code.
- Cooperative code must run to completion before other cooperatively scheduled code can run.



3. Contiki OS

3. Protothreads

Motivation

- Event handlers cannot execute blocking wait due to run-to-completion semantics in case of a single shared stack.
- Disadvantage: State machines become complex and cannot be expressed by a single event handler
  → difficult to program and maintain

New concept of lightweight threads: **Protothreads**

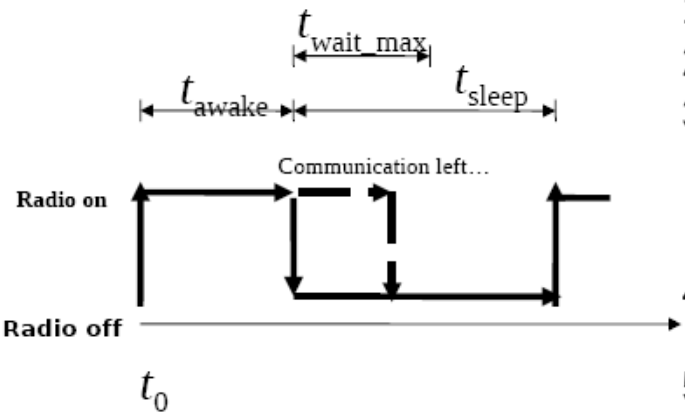
= memory-efficient programming abstraction that shares features of both multi-threading and event-driven programming to achieve low memory overhead.

- combine events and threads (blocking event handlers)
- provide several conditional blocking wait statements: e.g., PROCESS_WAIT_EVENT_UNTIL() blocks until conditional statement becomes true
  → linear sequencing of statements in event-driven programs
- are stackless: All Protothreads share the same stack, which is rewound every time a Protothread blocks. Local variables are not preserved across blocking wait statements.
- are cooperatively scheduled. Protothreads must always explicitly yield control back to kernel.

3. Contiki OS

3.1 Example: Simple MAC Protocol

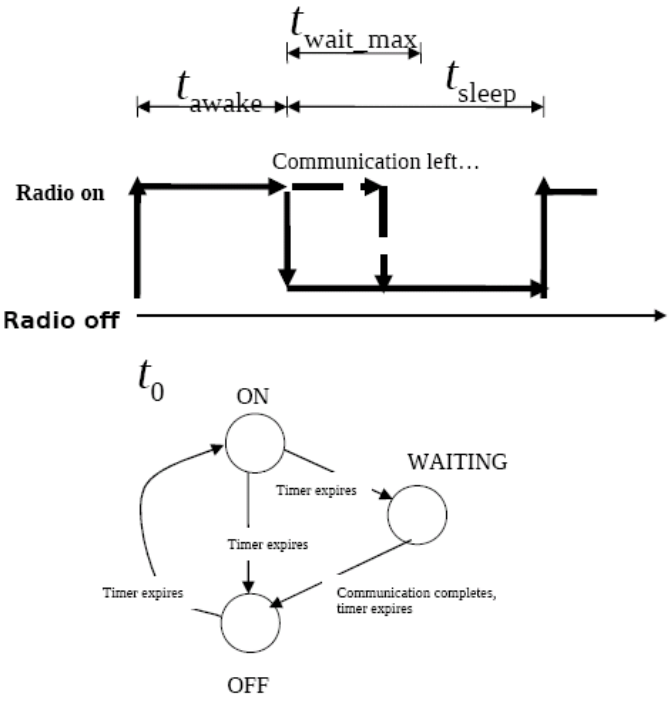
T-MAC like MAC protocol: switching on / off the radio at scheduled intervals.



1. Turn radio on.
2. Wait until $t = t\_0 + t\_awake$.
3. If communication has not completed, wait until it has completed or $t = t\_0 + t\_awake + t\_wait\_max$.
4. Turn the radio off. Wait until $t = t\_0 + t\_awake + t\_sleep$.
5. Repeat from step 1.

3. Contiki OS

3.2 Simple MAC Protocol with State Machine




```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
```

3. Contiki OS

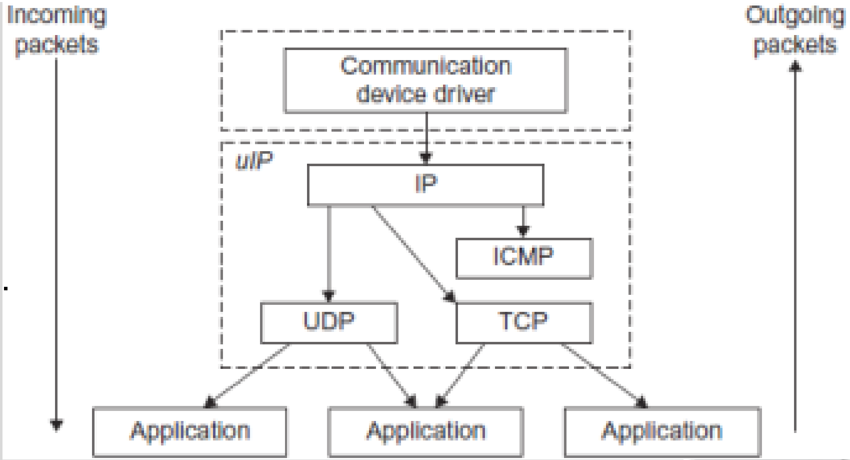
3.3 Simple MAC Protocol with Protothreads

```
static struct etimer timer;
static struct etimer wait_timer;
PROCESS_THREAD(radio_wake_process, ev, data)
{
   PROCESS_BEGIN();
   while(1) {
          NETSTACK_RADIO.on();
          etimer_set(&timer, T_AWAKE);
          PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));
          etimer_set(&timer, T_SLEEP);
          if(!NETSTACK_RADIO.channel_clear()) {
                 etimer_set(&timer, T_WAIT_MAX);
                 PROCESS_WAIT_EVENT_UNTIL(NETSTACK_RADIO.channel_clear() ||
                      etimer_expired(&wait_timer));}
          NETSTACK_RADIO.off();
          PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));}
   PROCESS_END();
}
```

3. Contiki OS

4. uIP Stack

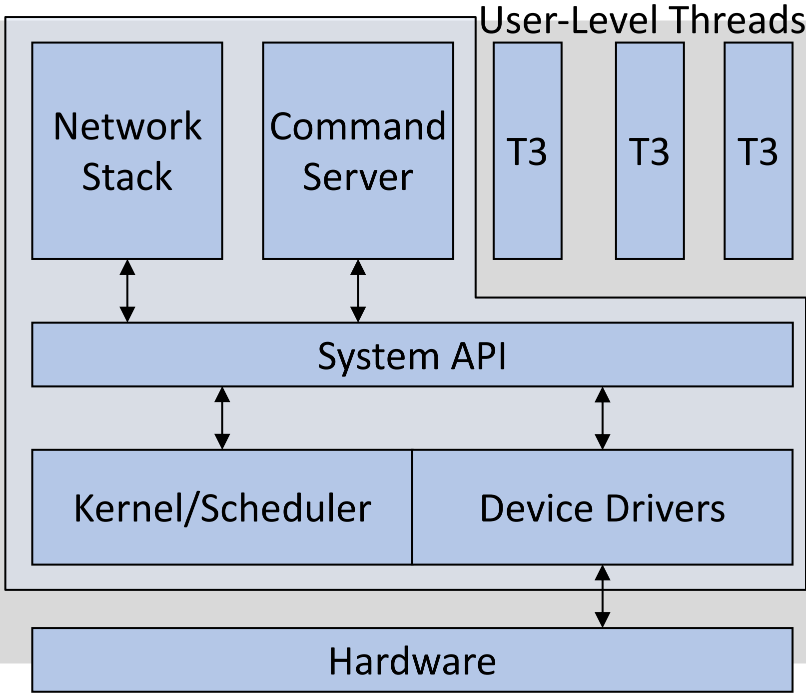
- small full TCP/IP stack supporting IPv4 and IPv6
- open source
- RFC-compliant to a large extent!
- intended for tiny microcontroller systems, where code size and RAM are severely constrained.
- uIP requires 4-5 KB code space and a few 100 bytes RAM.
- may eat up large share of memory of TelosB, only use when needed!
- limitations to minimize resource usage
  - Only one packet buffer for sending & receiving
  - Only one segment “in flight” at a time
    - → congestion window = 1
    - → no need for congestion control



4. MANTIS OS

1. Overview

- Classical layered multi-threaded operating system structure including
  - Multi-threading
  - Pre-emptive scheduling with time slicing
  - I/O synchronization via mutual exclusion
  - Network stack
  - Device drivers
- MANTIS code size
  - 14 KB flash memory
  - 500 bytes RAM



4. MANTIS OS

2. Memory and Thread Management

- Memory management
  - Allocation of space for global variables at compile time
  - Rest of memory managed as heap. Stack for each thread is allocated from heap.
- Thread management
  - Static, fixed size thread table
  - Semaphore support for applications
- Interrupts
  - Timer interrupts are handled by kernel.
  - Hardware interrupts are sent to device drivers.
  - No software interrupts
- Network stack options
  - Different layers (application, network, MAC, physical) can be implemented in different threads.
  - All layers can be implemented in a single thread.

4. MANTIS OS

3. Power Management and Dynamic Reprogramming

Power Management

- `sleep(period)` enables power-save mode.
- If all threads sleep: OS determines earliest deadline and shuts down microcontroller.

Dynamic Reprogramming

- Reprogramming granularities
  - Re-flashing of entire OS
  - Reprogramming single threads
  - Changing of variables within a thread
- Implementation as system call library
  - Application (e.g. command server) may write a new code image through call to system call library into EEPROM.
  - Application calls `commit` operation, which writes a control block for the boot loader.
  - Control block causes installation of the new code on reset.

UNIVERSITÄT BERN

5. RIOT

1. Overview

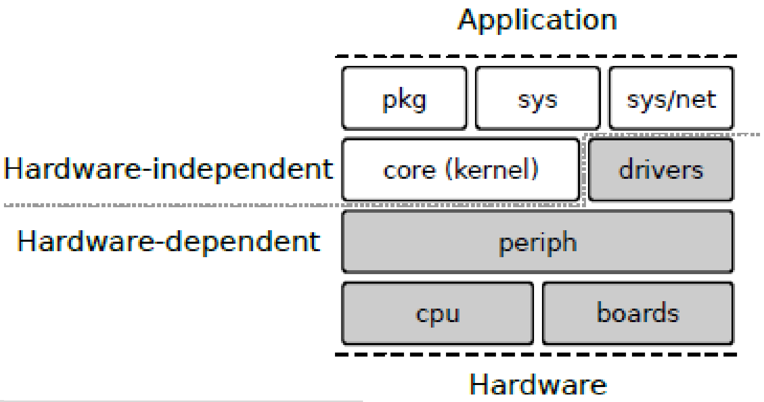
- Microkernel architecture supporting multi-threading, including mutual exclusion
- Static memory allocation to guarantee run-times
- supports various C++ libraries
- TCP/IP network stack (6LowPAN, RPL, IPv6, TCP, UDP, CoAP) and Named Data Networking
- supports several MCUs such as a 16-bit MSP430 or a 32-bit ARM7

memory requirements:

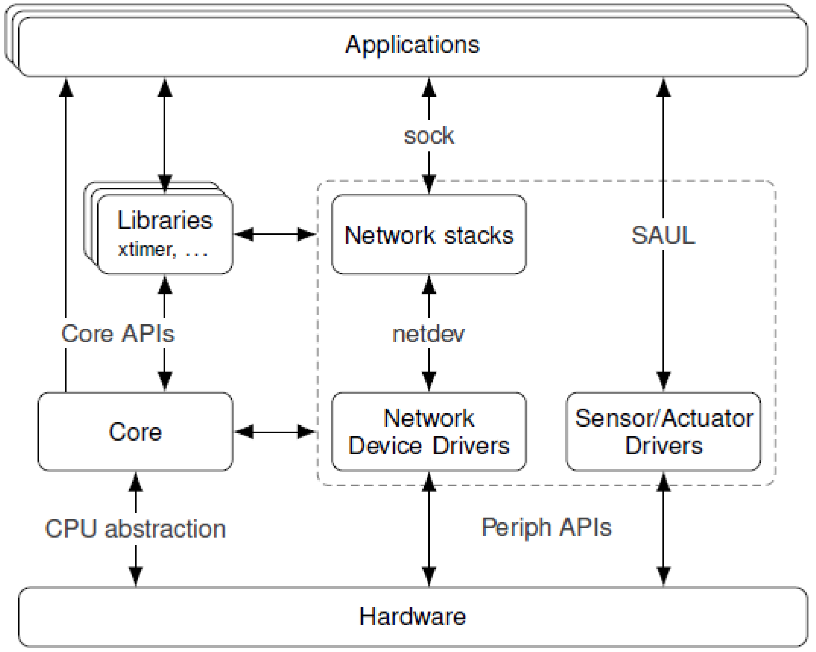
RIOT Configuration	ROM	RAM
Basic RTOS	3.2 kB	2.8 kB
6LoWPAN-enabled	38.5 kB	10.0 kB
JavaScript-enabled	166.2 kB	29.1 kB
OTA-enabled	111 kB	17.5 kB

5. RIOT

2. Architecture



- core: OS kernel
- Hardware abstraction
  i. CPU: microcontroller functionality
  ii. boards: configures CPU and drivers
  iii. drivers: HW independent device drivers for sensors, actuators, network transceivers, storage components with high-level APIs
  iv. periph: unified access to microcontroller peripherals
- sys: system libraries
- pkg: 3rd party components



5. RIOT

3. Kernel

Multi-threading

- overhead by thread control block (36 bytes), stack, CPU context < 128 bytes
- Light-weight inter-process communication using semaphores, mutex, messaging

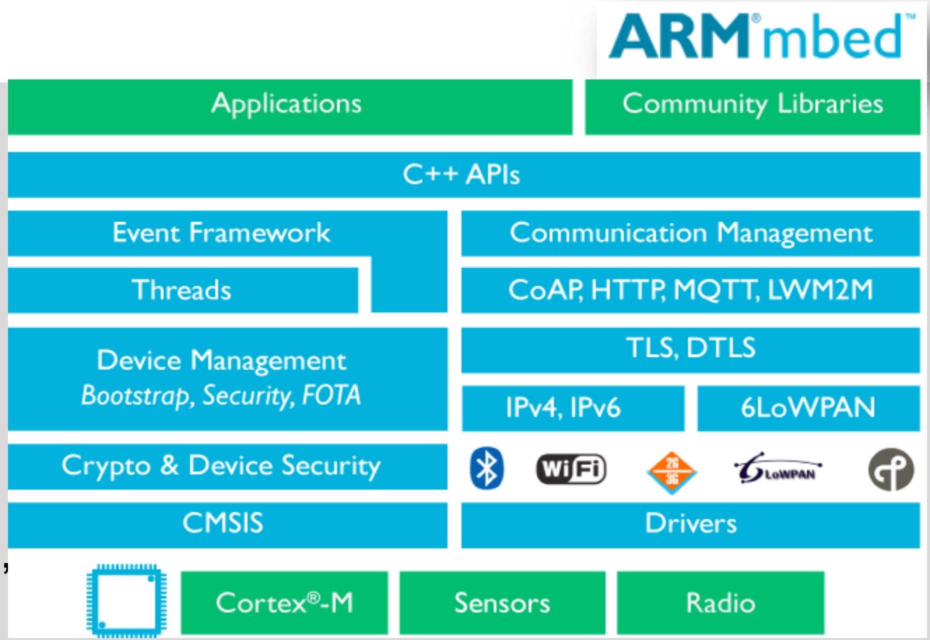
Tickless O(1) Priority Scheduler

- works without periodic events.
- Whenever there are no pending tasks, RIOT will switch to idle thread.
- The only function of the idle thread is to determine the deepest possible sleep mode, depending on the peripheral devices in use.
- Only interrupts (external or kernel-generated) wake up the system from idle state.

6. Other IoT Operating Systems

1. mbed

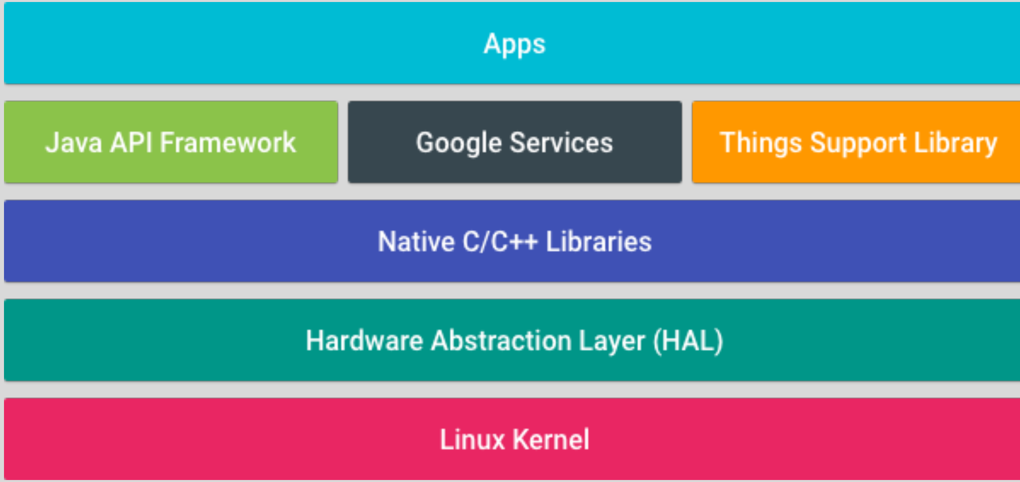
- Chip designers are developing "semi-open" OS platforms, e.g., ARM Microsystems has created mbed for 32-bit ARM Cortex-M microcontrollers.
- C/C++ based application development platform, online compilation in the cloud
- Tools for creating microcontroller firmware
- Core libraries providing peripheral drivers, networking, runtime environment, build tools, test and debug scripts.
- Components database provides driver libraries for components and services.



6. Other IoT Operating Systems

2. Android Things

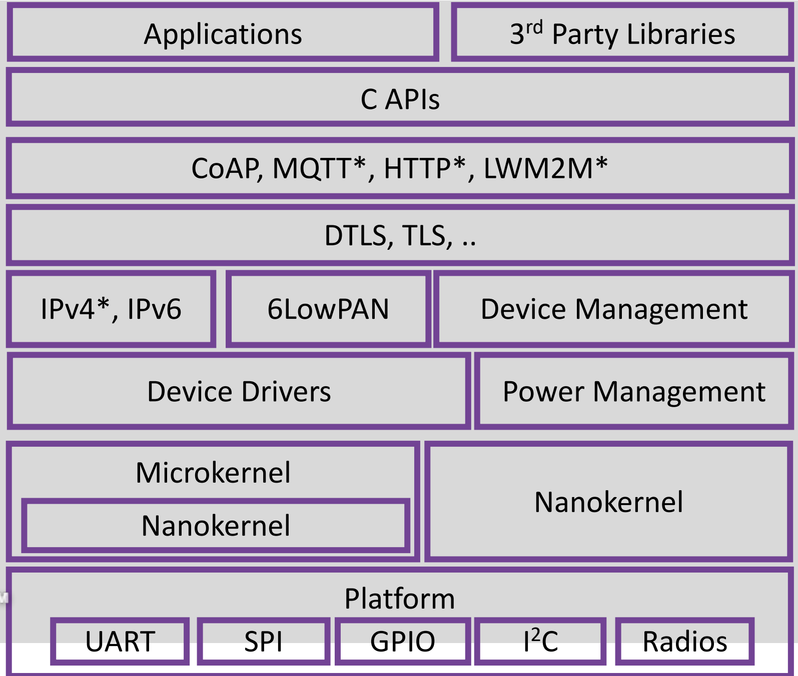
- Android based embedded OS designed to work with 32-64 MB RAM
- Supporting BLE and WiFi
- Software Development Kit for application development, similar to developing Android applications



6. Other IoT Operating Systems

3. Zephyr

- Linux Foundation hosted collaboration project.
- producing open source code
- small, scalable, library-based real-time operating system, optimized for resource constrained devices across multiple architectures
- supports hardware > 8 KB RAM
- 1 single executable executed in 1 single address space
- Multi-threading support, inter-thread communication
- Kernel mode only, no user-space, no dynamic runtimes
  - Nanokernel: Limited functionality targeting small memory footprint (below 10kB RAM)
  - Microkernel (superset of nanokernel): with additional functionality and features
- Memory and resources are typically statically allocated.



UNIVERSITÄT BERN

6. Other IoT Operating Systems

4. FreeRTOS

- Micro kernel OS
  - Reduced kernel size (3 C files)
  - All other functions are provided by servers running on user level.
- Preemptive priority-based round-robin scheduling
- Multi-threading support
- Idle task puts microcontroller into low power mode
- Event-based memory allocation
- Third-party network support, e.g. uIP

7. Operating System Power Management

UNIVERSITÄT BERN

Dynamic Power Management

- CPU-centric
  - Low-Energy Earliest Deadline First Scheduling
- I/O-centric
  - Low-Energy DEvice Scheduling

7. Operating System Power Management

1. Low-Energy Earliest Deadline First

t_c: current time;
$S_h > S_{l1} > S_{l2} > \ldots S_{lm}$: Available processor speeds

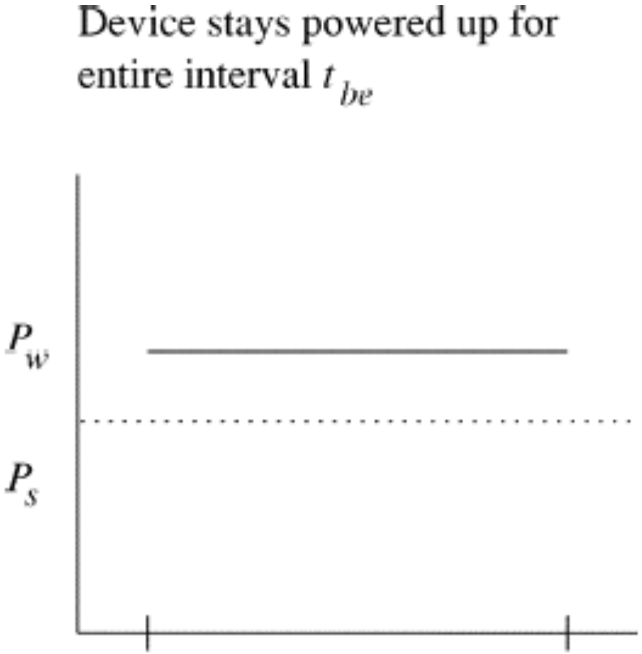
```
schedulable = 1
if ready_list ≠ NULL
    Sort task deadlines in ascending order;
    Select task τ_i with earliest deadline;
    for S = S_lm to S_h
        if t_c + l_i/S ≤ d_i then
            t = t_c + l_i/S
            for each task τ_u that has not completed execution
                if t + l_u/S_h ≤ d_u then
                    t = t + l_u/S_h
                else
                    schedulable = 0
                    break
            endfor
            if schedulable = 1 then
                Schedule τ_i at S
                break
            endif
endfor
```

d_i: deadline for task τ_i

l_i: length of task τ_i (instruction cycles)

7. Operating System Power Management

2.1 Low-Energy DEvice Scheduling

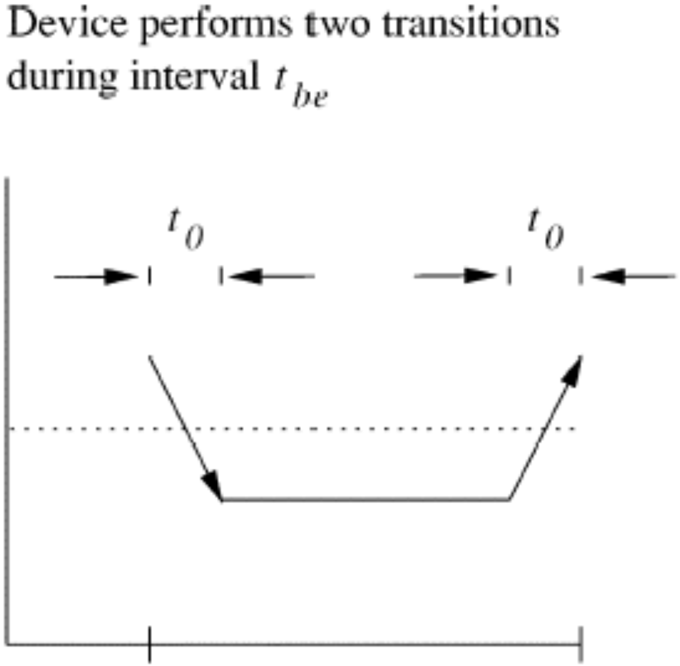


Set of tasks with

- start times s_i
- completion times c_i

$t_{0,j}$: state transition time for device k_j,

L_i: devices needed by task i



6. Operating System Power Management

2.2 LEDES

Algorithm LEDES(k_j, τ_i, τ_{i+1})

`curr`: current scheduling instant;

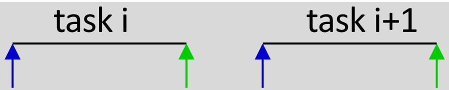
- **if** $\texttt{curr} = s_i$ — begin of task i
  - **if** k_j is powered-up
    - **if** $k_j \notin L_i \bigcup L_{i+1}$
      - **shutdown** k_j

    shut down device if not used by next 2 tasks (i, i+1)
  - **if** $k_j \in L_{i+1}$ and
    - **if** $s_{i+1} - (s_i + c_i) \geq t_{0,j}$
      - **shutdown** k_j

    shut down device for task i+1 if there is enough time for starting device between tasks i and i+1
  - if (k_j is shut down)
    - **if** $k_j \in L_{i+1}$ and $s_{i+1} - (s_i + c_i) < t_{0,j}$
      - **wakeup** k_j

    start device for task i+1 if there is not enough time for starting device between tasks i and i+1
- **if** $\texttt{curr} = s_i + c_i$ — end of task i
  - **if** k_j is powered-up
    - **if** $k_j \notin L_{i+1}$ and $s_{i+1} - \texttt{curr} \geq t_{0,j}$
      - **shutdown** k_j

    shut down device if not needed for task i+1 and if there remains enough time to shut down before task i+1 starts, because it may be necessary to wake the device up again then.
  - **else** if ($k_j \in L_{i+1}$)
    - **wakeup** k_j



Thanks

for Your Attention

UNIVERSITÄT
BERN

Prof. Dr. Torsten Braun, Institut für Informatik

Bern, 08.03.2021