Embedded Information Systems

7. Embedded operating systems

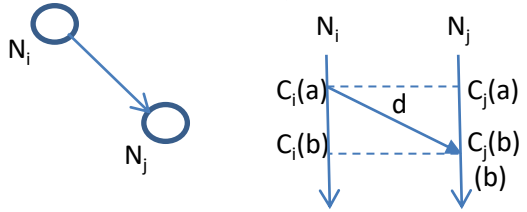
Sensor networks

November 24, 2020

Synchronization techniques: Taking one sample

Reminder:

Unidirectional Synchronization:



Node N_j is not familiar with d, its knowledge is only the fact, that the clock of node N_i displayed the value $C_i(a)$ before the clock of node N_j displayed $C_j(b)$.

To perform synchronization, we must estimate either the value of $C_j(a)$ or $C_i(b)$.

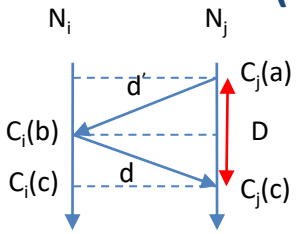
If the limits $d_{min} \leq d \leq d_{max}$ are known, then

$$\hat{C}_j(a) \approx C_j(b) - \frac{d_{min} + d_{max}}{2}$$ or $$\hat{C}_i(b) \approx C_i(a) + \frac{d_{min} + d_{max}}{2}$$

Having these estimates, the clock of node N_j should be modified by $\hat{C}_j(a) - C_i(a)$ or $C_j(b) - \hat{C}_i(b)$.

If the communication jitter $(d_{max} - d_{min})$ is large, then the synchronization will be inaccurate, because the lower bound of $C_j(a)$ will be $C_j(b) - d_{max}$, and the upper bound will be $C_j(b) - d_{min}$, which is a wide range.

Bidirectional (round trip) synchronization:



Here node N_j knows that $0 \leq d \leq D$. $D = C_j(c) - C_j(a)$. If $d_{min} \leq d \leq d_{max}$, then $\max(D - d_{max}, d_{min})$ and $\min(d_{max}, D - d_{min})$ give the limits of d.

The estimate that can be computed here:

$$\hat{C}_j(b) \approx C_j(c) - \frac{D}{2},$$

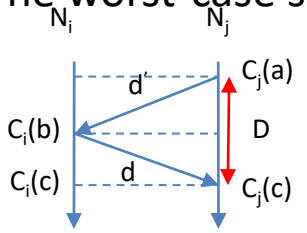
having lower bound $C_j(c) - (D - d_{min})$, and upper bound $C_j(c) - d_{min}$.

The clock of node N_j is to be modified by

$$\hat{C}_j(b) - C_i(b).$$

With such a method the quality of the synchronization will be better.

The worst-case synchronization error: $\frac{D}{2} - d_{min}$, that can be proved using the figure.

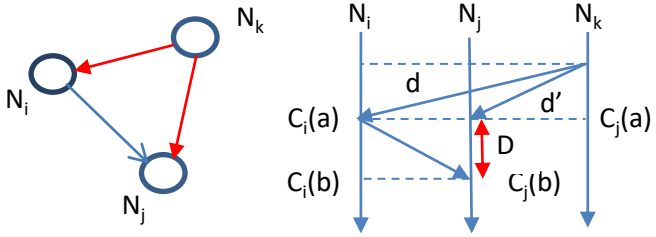


Reminder:

The method can be improved using the so-called probabilistic time synchronization, where node N_j after receiving the timestamp checks weather the value of $\frac{D}{2} - d_{min}$ < than a defined threshold. If not, then the request will be repeated.

Reference broadcasting synchronization:

In this case also a so-called *beacon* node N_k is involved. The beacon sends a broadcast message to the other nodes.



The delays are almost equal:

$d \approx d'.$ Thus $\hat{C}_i(b) \approx C_i(a) + D,$

i.e., the clock of node N_j should be modified by $C_j(b) - \hat{C}_i(b)$.

It is an important property, that the synchronization of node N_j is performed without using it radio channel.

The accuracy degrades with the hop distance from the root!

Synchronization of multiple nodes:

(1) Single-hop synchronization with a set of master nodes which are synchronized e.g. using GPS;

(2) Partitioning the network into clusters: all nodes within a cluster can broadcast messages to all other members of the cluster and thus reference-broadcast techniques can be used to synchronize the cluster internally. Some nodes are members of several clusters and participate independently in all corresponding synchronization procedures. These nodes act as time gateways to translate time stamps from one cluster to the other;

(3) Tree construction: The most common solution of the multi-hop synchronization. Problem is to construct a synchronization tree with a single master at the root.

7. Embedded operating systems

Software aspects of Embedded Systems: Typical software architectures

Concerns: computational capacity, memory size (RAM, ROM), development, flexibility, reaction time to external, asynchronous event, (memory) protection, recursion, re-entrant calls, processor utilization, etc.

Features used in characterization:
- maximum response time,
- handling hardware,
- inter task communication,
- design technologies,
- application area.

Classification of software architectures:
- periodic,
- priority-based,
- event-triggered,
- time-triggered.

Practical implementations:
- cyclic architecture
- cyclic architecture together with external interrupt
- function queue scheduling
- RTOS

Cyclic architectures:
- round-robin
- weighted round-robin
- time-triggered round-robin
- strictly time-triggered
- round-robin with interrupt

Properties:
- maximum response time: tA+ tB+ tC+ ..., i.e. maximum cycle time.
- hardware is handled with polling
- inter-task communication with shared variables (non preemptive!)

Simple cyclic architecture: the processor runs in an infinite loop, even if there is no service request.

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceC_Needs_Service()) {Service_C};
...
}
```

- development: hard
- hard RT behaviour: slow (e.g. printer task) (however, it can be RT)
- processor utilization:100% (this is wrong!)
- application area: where the time constant of the system is larger than the cycle time (fast and rare events).

Weighted round-robin: the more frequent tasks can be called more than once within the cycle.

```
void main() {
while (TRUE){
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceB_Needs_Service()) {Service_B};
if (DeviceA_Needs_Service()) {Service_A};
if (DeviceC_Needs_Service()) {Service_C};
if (DeviceA_Needs_Service()) {Service_A};
...
}
}
```

Properties:

- maximum response time: tA+ tB+ tA+ tC+ tA+.., but for more frequent tasks it is less than the maximum cycle time.
- hardware is handled with polling
- inter-task communication with shared variables (non preemptive!)
- development: hard
- processor utilization:100% (this is wrong!)
- further property: priority-like behaviour, still not preemptive

Time-triggered round-robin:

The boundaries of the cycle are determined by a timer. For every timer interrupt the cycle runs once or several times. The cycle itself can be weighted round-robin.

Properties:

- maximum response time: cycle period.
- hardware is handled with polling
- inter-task communication with shared variables
- development: hard
- hard RT behaviour: slow (e.g. printer task) (however, it can be RT)
- processor utilization: <100% (standby exists)

Strictly time-triggered:

Every task starts running at predefined time instant.

Administration: the predefined time instants and function references are collected into a table, which is used by a micro run-time system to start the scheduled task at right time.

Properties:

- maximum response time: the scheduling rate of the task + its computation time
- inter-task communication with shared variables

Properties of Strictly time-triggered (cont.):

- development: hard
- processor utilization: <100% (standby exists)
- hard real-time behaviour: OK, its application is typical in safety-critical systems

Round-robin with interrupt:

Signalling is performed with interrupt instead of polling.

```
FLAG A, B, C;
void interrupt A_Handler() { Handle_HW_A(); A=TRUE; }
void interrupt B_Handler() { Handle_HW_B(); B=TRUE; }
void interrupt C_Handler() { Handle_HW_C(); C=TRUE; }
void main() {
while (TRUE){
 if A {A=FALSE; Service_A(); }
 if B { B=FALSE; Service_B(); }
 if C { C=FALSE; Service_C(); }
...
}
}
```

Properties:

- maximum response time: tA+ tB+ tC+ ...(+ IT) only the signalling will be faster, the service not.
- hardware is handled with interrupt, priority assignment to interrupts is possible.
- inter-task communication with shared variables: no problem. Between interrupt and task: pre-emption (shared variables may cause problems).
- development: concerning interrupts it is good, but by introducing new tasks time conditions will change
- application field: if the execution time of the tasks is nearly the same.

This is the most widely used solution.

Function-Queue Scheduling

```
void interrupt A_Handler() { Handle_HW_A();
PutFunction(Service_A); }
void interrupt B_Handler() { Handle_HW_B();
PutFunction(Service_B); }
void interrupt C_Handler() { Handle_HW_C();
PutFunction(Service_C); }
void Service_A();
void Service_B();
void Service_C();
void main() {
        while (TRUE){
                while (IsFunctionQueueEmpty());
                        CallFirstFromQueue();
        }
}
```

Properties:

- maximum response time: execution time of the longest task + the execution time of the *i*th task.
- hardware is handled with interrupt
- inter-task communication with shared variables: no problem. Between interrupt and task: pre-emption (shared variables may cause problems)
- development: easy
- The service order from the queue can be: (1) FIFO, (2) priority based
- drawback: non preemptive
- processor utilization:100%

Question: how to modify to have less than 100%?

Software based on Real-time Operating System

Properties:

- maximum response time:
  a feature of the operating system (~10 µsec) + the execution time of the task together with the sum of the execution time of higher priority tasks
- hardware is handled with interrupt
- inter-task communication:
  using RTOS communication functions.
  This solves synchronization, as well.
- development: easy
- hard real-time behaviour: good

```
void interrupt A_Handler() { Handle_HW_A(); Signal_A(); }
void interrupt B_Handler() { Handle_HW_B(); Signal_B(); }
void Service_A();
void Service_B();
void task_A(void) {
        while (TRUE){
                Wait_for_Signal_A(); Service_A();
        }
}
void task_B(void) {
        while (TRUE){
                Wait_for_Signal_B(); Service_B();
        }
}
```

- processor utilization:<100% (In idle state sleep is possible)
- application area: everywhere
- drawback: The Operating System (OS) involves additional code and time

Some terms:

embedded OS:	small resource requirement (micro-controllers (µC) are enough)
real-time OS:	provides finite, deterministic response time
task:	series of inter-dependent activities
job:	parts, subunits of the tasks
process:	unit of schedule with separate memory block (implementation of tasks)
thread:	unit of schedule without separate memory block
kernel:	the key components of the OS
scalability:	the services of the OS can be switched on/off in compilation time; availability with source code

The role of the kernel:

- to provide parallel programming environment,
- scheduling,
- inter-task communication,
- handling interrupts,
- handling timers, and timing services,
- (possibly) memory management

Further OS features:

- handling peripherals and system programs (APIs: Application Programming Interfaces)
- handling communication channels
- virtual memory management, file system, etc.

Comparison of desk-top OSs and embedded OS-s

a. Desk-top OS-s are not suitable in embedded systems, because:

(1) the services are too extensive; (2) non-modular, non-fault tolerant, non-configurable, non-modifiable; (3) require large memory; (4) not optimized for power consumption; (5) are not designed for mission-critical application; (6) timing uncertainties are too large.

b. Configurability is needed because:

- a single OS is unable to meet all the requirements;
- the overhead caused by the unused functions and data is not tolerable;
- there are many embedded systems not having disc, keyboard, display, mouse.

Typical tools of configuration:

- removal of the unnecessary functions e.g. by the linker;
- applying conditional compilation (using #if and #ifdef commands);

Comment:

The verification of systems, having operating systems generated by configuration, is difficult:

- every OS generated by configuration should be thoroughly tested;
- E.g. the number of the configuration points of the OS eCos (an open source RT OS of Red Hat) is between 100 and 200.

c. The device handlers of the embedded OS are handled by the tasks, and not by the integrated drivers;

- the predictability is improved, if everything is handled by the scheduler;
- practically there is no such a device, which would be supported by all versions of the OS, apart from the timer.

Embedded RTOS	Standard OS
application software	application software
middleware	middleware
device drivers	OS
real-time kernel	device drivers

d. In embedded systems **every task can use interrupt**:

- In standard OS this would be a serious source of unreliability;
- Embedded programs are supposed to be tested;
- It is allowable that an interrupt starts or stops tasks by putting the start addresses of the tasks into the interrupt table. This is more efficient and predictable than via OS functions.

Comment: However, composability will fail: if the run of a task depends on an interrupt, then it is difficult to add another task to be started by the very same event.

- If RT processing is a concern, then the time required by the interrupt services should also be considered. In this case the interrupts should also be handled by the scheduler.

e. In embedded operating systems **the protecting mechanisms are not necessary** in every case:

- The embedded systems are designed for dedicated purposes, untested programs are rarely used, the software is reliable.
- There is no need for privileged I/O instructions, the tasks can manage the I/O operations related to them.
- However, in case of security concerns protecting mechanisms might be required.

f. The real-time operating systems (RTOS) support creating real-time systems. Requirements:

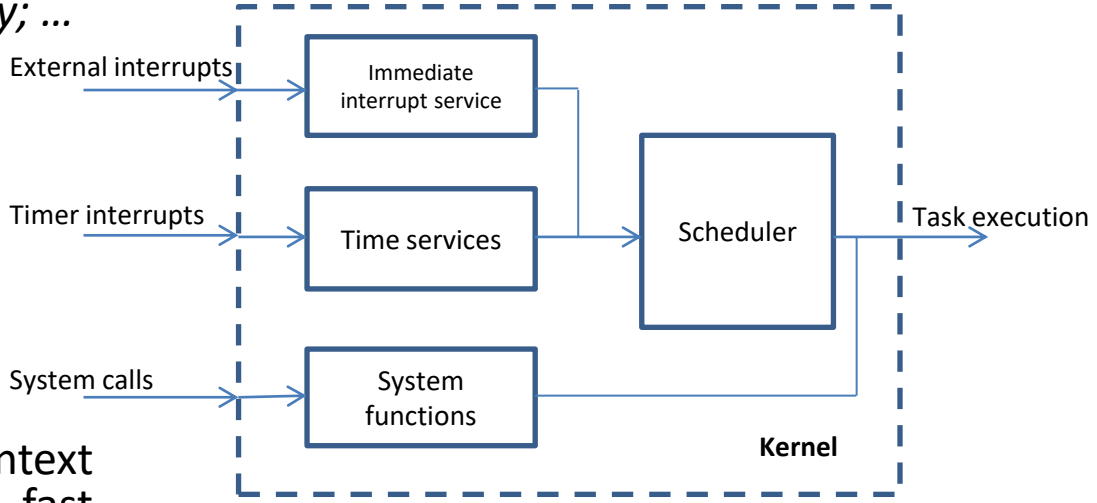
- The time-domain behaviour is predictable: the maximum execution time of every operating system function should be known. The RTOS behaves in a deterministic way, almost all activity is supervised by the scheduler.
- The RTOS manages timing and scheduling: to do this it should familiar with deadlines of the tasks, and should provide high-resolution timing services.
- The RTOS should be fast (due to practical considerations).
- The RTOS should provide process management functions, so-called Application Program Interface (API), like:

create_thread, suspend_thread, destroy_thread; …
create_timer, timer_sleep, timer_notify; …
open, read; …
other system calls.

The role of the kernel:

Execution of concurrent (quasi-parallel) programs in forms of tasks or threads:

- by handling the states of tasks/threads, and by ordering them into queues;
- by executing preemptions (fast context switching, see Figure below) and fast interrupt handling;

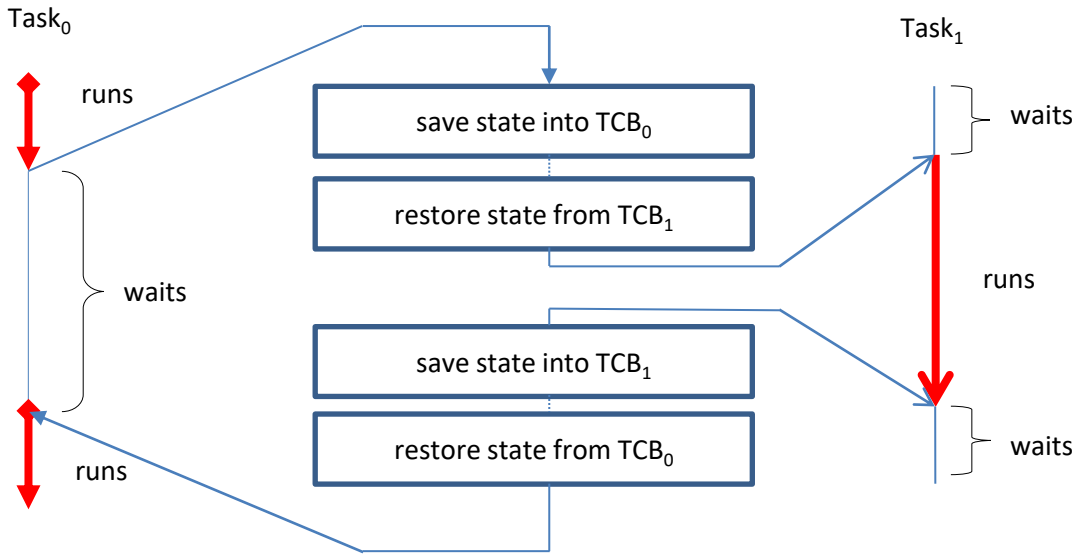


Scheduling the CPU (guaranteeing deadlines, minimizing waiting, reasonable distribution of computing power);

Synchronization of tasks (Critical sections, semaphores, monitors, mutual exclusion);

Inter-task communication (buffering);

Supporting RT clock serve as internal reference.



TCB: task control block: contains the run-time environment (context, state) of a task. While it is running, it is typically loaded into the registers of the processor, and saved into the memory in case of pre-emption.

g. Real-time extensions of standard operating systems

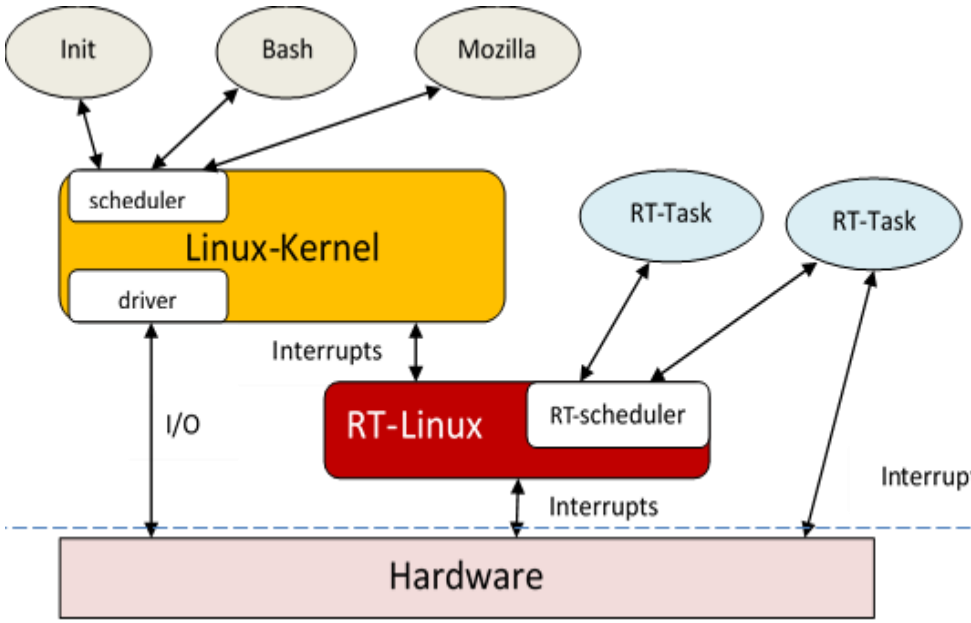
A possible solution: Every real-time task is executed by a real-time kernel, and the standard operating system is a single task:

RT-task 1	RT-task 2	non-RT task 1	non-RT task 2
device driver	device driver	Standard-OS	
real-time kernel			

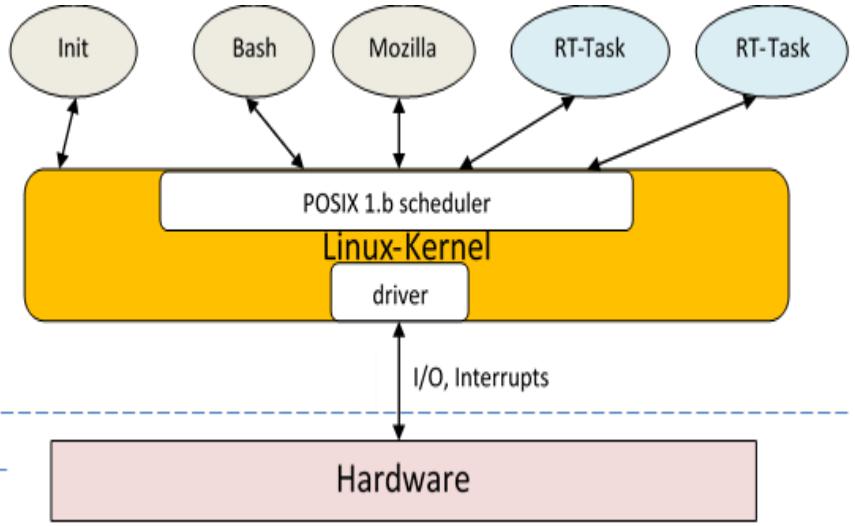
Comments:

- Problems within the standard OS do not influence the execution of the real-time tasks.
- Since the real-time tasks are unable to use the standard OS function this solution is below the expectations.

Example: RT Linux



Example: Posix 1.b RT-extensions to Linux



The ordinary Linux scheduler can be replaced by the POSIX scheduler, which provides priority for the real-time tasks.

Among the standard OS functions special real-time functions are also offered.
Programming is simple, however, there is no guarantee to meet the deadlines.

(POSIX: "**P**ortable **O**perating **S**ystem **I**nterface for uni**X**".)

Virtualization in embedded systems

Virtualization: supports portability, i.e. the use of software on different hardware platforms.
A so-called **virtual machine** (VM) provides such a software environment for the given software, like it would run on a real hardware in the following structure:

Application
Operating system
Hypervisor
Processor

The software layer, which provides the virtual environment is the so-called **virtual machine monitor** (VMM) or **hypervisor.** It has three key features:

- provides an identical software environment than the original machine;
- at maximum it runs slower;
- completely supervises the system resources.

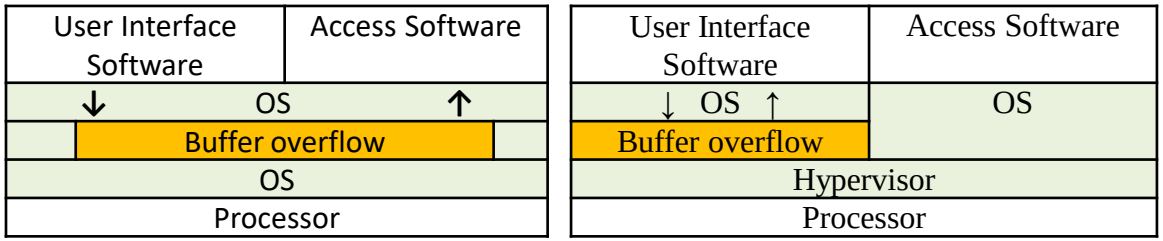
The majority of the VM instructions are immediately executable on the hardware applied, some of them requires interpretation. Among them there are:

- control-sensitive instructions, which modify the privileged machine states, therefore interfere with the supervision of the resources by the hypervisor.
- behaviour-sensitive instructions, which can reach (read) the privileged machine-states.

Concurrent operating systems on virtual machines

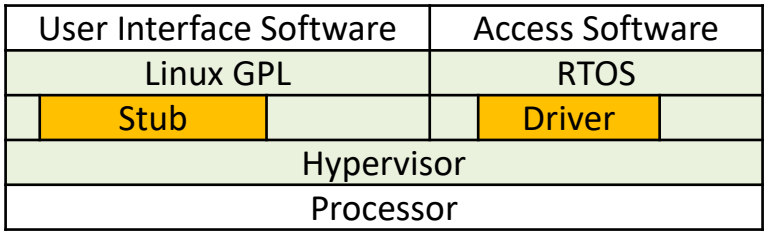
User Interface Software	Access Software
Standard OS	RTOS
Hypervisor	
Processor	

Improvement of security using virtualization



The error caused by an application does not propagate towards another, because it has its own operating system.

Licence separation using virtualization



GPL: **General Public Licence.**

Linux is licenced under the GPL which requires open-sourcing of all delivered code.
Therefore, all software written using Linux is **open-source.**

Where this would cause problems, the Linux and the dedicated application run on different virtual machines. A **stub** (or **proxy**) **driver** is used to forward Linux driver requests to the real device driver, using **hypercalls.**

Limits of virtualization in embedded systems:

- Using more operating systems together with applications of increasing complexity results in large amount of code, which can be a source of error, requires larger memory, consumes more energy.
- The different subsystems should intensively cooperate: separated implementation does not fit.
- The inter-subsystem communication is not supported by the virtual machine.
- To share common resources among subsystems is hard to organize if more operating systems are running parallel.
- Due to the virtualization scheduling is performed at two levels:
  (1) Between the hypervisor and the VM,
  (2) within every operating system running on the VM.
- The fulfilment of critical security requirements is not supported by the virtualization alone. The critical code segments (so-called trusted computing base, TCB) must be executed in privileged mode on the processor.The hypervisor is part of the TCB.
  The correctness of such a code should be proved.
- Virtualization increases the size of the code.

What kind of supporting software is required by the embedded?

- support for virtualization with all its benefits;
- support for lightweight but strong encapsulation of medium-grain components that interact strongly, to build robust systems that can recover from faults;
- high-bandwidth, low-latency communication, subject to configurable, system-wide security policy;
- global scheduling policies interleaving scheduling policies of threads from different subsystems;
- ability to build subsystems with very small trusted computing base (TCB).

Microkernel (microvisor) technology: a better solution to embedded systems

A microkernel (microvisor) is a minimal privileged software layer that provides only general mechanisms.

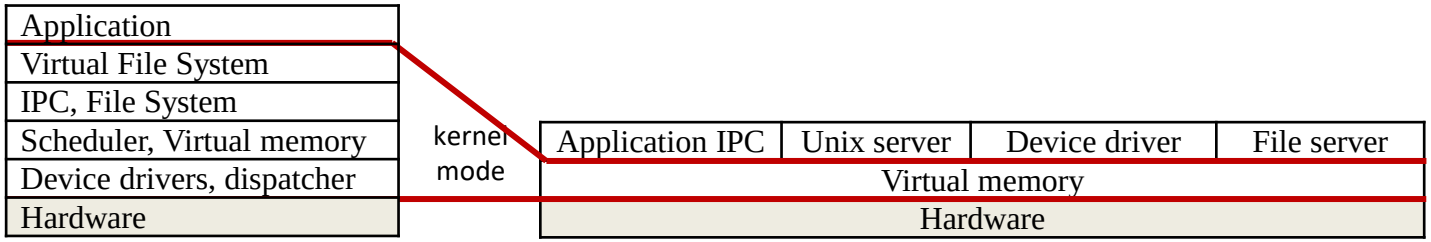
Actual system services and policies are implemented on top in user-mode components.

A microkernel is defined by Liedtke's minimalism principle (1995):

A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.

This minimality implies that a microkernel does not offer any services, only the mechanisms for implementing services.

The microkernel approach leads to a system structure that differs significantly from that of classical "monolithic" operating systems:



The classical structures have a vertical structure of layers, each abstracting the layers below, a microkernel-based system exhibits a horizontal structure.
System components run beside application code and are invoked by sending messages.
In case of a microkernel system there is no real difference between “system services” and “applications”, all are simply processes running in user mode.
Each such user-mode process is encapsulated in its own hardware address space, set up by the kernel.
It can only affect other parts of the systems (outside its own address space) by invoking kernel mechanisms, particularly message passing (Inter Process Communication, IPC).
It can only directly access memory or other resources if they are mapped into its address space via a system call.

A more detailed description about microkernels, e.g.:
https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products/

Sensor networks

The availability of cheap sensors to measure almost all possible quantities resulted in systems consisting (sometimes of large amount) of nodes capable to measure, pre-process and communicate data coming from the environment.

A typical example of such a node is the **Berkeley Mica2 mote** (see picture).
Its size can be estimated by the size of **two AA type batteries** located underneath the printed circuit board.

Concerning this type of devices, very many information is available on the internet.

The sensor network applications can be characterized by relatively
(1) large spatial scope, (2)large number of nodes, and (3) limited availability of local power.

Examples of applications:

1. Detection of acoustic shockwaves. Shooter localization → Counter-sniper system
2. Detection of shooting at elephants → help authorities to catch poachers
3. Intelligent rock bolt monitoring
4. Active noise control

(A detailed slide-set is available on the webpage of the subject. Plenty of similar presentations including many references can be found on the internet.)

The TinyOS operating system

Why it is needed?

Traditional operating systems have difficulties in case of sensor networks, because multi-threaded architecture cannot be used with sufficient efficiency, they need large memory, and do not support the minimization of power consumption.

In case of sensor networks the followings are important:

(1) concurrent execution, (2) efficiency of energy utilization, (3) small memory footprint, and (4) the support of various utilization.

Major properties of TinyOS:

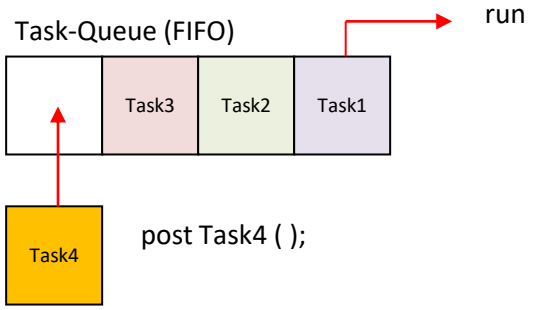
- The TinyOS is an open-source operating system, which was designed for sensor network applications.
- It is component-based, written in NesC (Networked embedded system C) language within a cooperation of University of California, Berkeley and Intel Research.
- The components-based architecture allows frequent changes while keeping the size of the code minimum.
- Program execution is event-based, hence support high concurrency.
- Power efficient, because the processor is sent into sleep mode as soon as possible.
- It has small footprint, because it applies a FIFO-based non-preemptable scheduling.
- TinyOS uses static memory allocation, memory requirements are determined at compile time. This increases runtime efficiency.
- Local variables are saved on the stack.
- Power-aware, two-level scheduling is applied: (1) Long running tasks and interrupt events, (2) Sleep unless tasks in queue, wakeup on event.
- Tasks are time-flexible, background jobs, atomic with respect to other tasks, can be pre-empted by events.
- Events are time-critical, shorter duration program sections, with LIFO (Last-in First-Out) semantic (no priority), can post tasks for deferred execution.
- Programs are built out of components, each component specifies an interface, interfaces are “hooks” for wiring components to result in a configuration.

- Components should use and provide bidirectional interfaces.
- Components should call and implement commands and signal and handle events.
- Components must handle events of used interfaces and also provide interfaces that must implement commands.

Component hierarchy:

- Command flow "downwards", they are non-blocking requests, and the control returns to the caller.
- Events flow upwards, post tasks (function queue scheduling), signal higher level events, and call lower level commands. The control returns to signaller.

To avoid cycles: events can call commands, commands can NOT signal events.



Communication in sensor networks

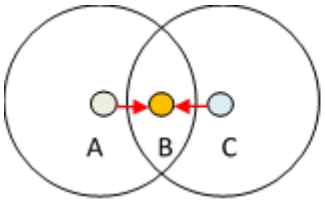
Standardized solutions: typically, using ISM (Industrial, Scientific, Medical) 2.4 GHz, spread spectrum: ZigBee/IEEE 802.15.4, IEEE 802.11b (Wi-Fi) WLAN (Wireless Local Area Network), Bluetooth WPAN (Wireless Personal Area Network).

Media Access Control: Dynamic (on demand): e.g. CSMA (Carrier Sense Multiple Access).

Collision avoidance: Before sending checks the channel. If information flow is detected: waits.

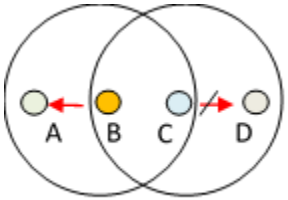
CSMA problems:

Problem of hidden terminal:



- A sends message to B
- C does not hear A!
- C also sends message to B
- B is unable to receive messages

Problem of visible terminal:



- B send message to A
- C would like to send message to D
- C hears B
- C does not send message, even if it would be possible

CSMA modifications

CSMA together with availability indication: two channels are used, one for data transmission, and another for indicating availability. The receiver continuously signalizes availability.

The sender before transmission checks both channels.

The node simultaneously sends and receives that is expensive.

Two simultaneous channels require wider bandwidth.

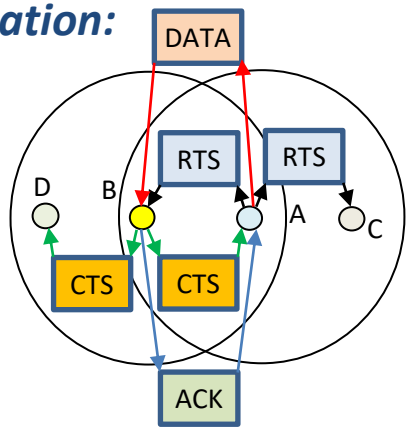
Request To Send/Clear To Send (RTS/CTS): Operates in two phases:

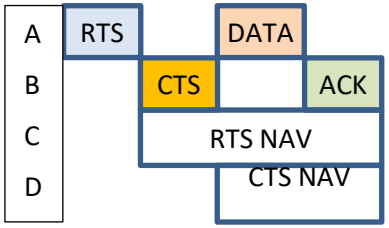
(1) Handshake, (2) Data transmission. The basic idea is: collision occurs at the receiver.

Avoids the hidden terminal problem.

It is advantageous in case of longer messages, otherwise the overhead is too large.

Its operation:





Sender „A" sends RTS message to "B"
Receiver „B" replies with CTS message
Receiving CTS the sender "A" transmits data

The other nodes are not allowed to transmit after receiving RTS or CTS!

(NAV = Network Allocation Vector)

Routing in sensor networks Sensor networks are ad-hoc.
Node distribution is random, connections come into being randomly, they are not reliable (fading), they can be mobile, and systems can consist of large number of nodes.

Typical configurations:

- Single source → multiple (possibly all) destinations.
  E.g. a central node propagates commands within the network.
- Multiple source → single destination. E.g. data collection and transmission into a central node.

- Single source → single destination. E.g. data exchange between nodes.

Data transmission strategies:

Time-triggered: Sensor activity and data transmission is time-triggered. E.g. data collection.
It is advantageous to conserve energy: sleep mode together with synchronized wakeup.

Event triggered: It is used in time-critical applications.
Sensor activity is due to some event in the environment.
The reduction of energy consumption is more difficult.

Polling: Sensors are activated by a command of a central node.

Typical network structures:

Flat: The nodes have equal rights; Scaling of the network is hard.

Hierarchical: Clusters are formed:
The communication is solved within the clusters and among the clusters separately.
Among the clusters the so-called control nodes communicate. They have extra capabilities.
The role of controlling a cluster can be dynamically changed.

Flat: *Flood routing:*

- Message distribution by *broadcasting*.
- The receiver after the first arrival of a message stores the message or only its identifier (if the destination of the message is another node), and broadcasts the message.
- *Typical application:* single source → multiple destination (command is distributed quasi simultaneously).
- *Its advantage:* simple, fault tolerant due to the high level of redundancy.
- *Its disadvantage:* very many superfluous message and energy consumption. In addition, collisions due to hidden terminals.

Modifications:

- the receiver broadcasts the message only with a probability *p*. The value *p* is topology dependent.
- to reduce collisions: after receiving the message random waiting time before transmission.

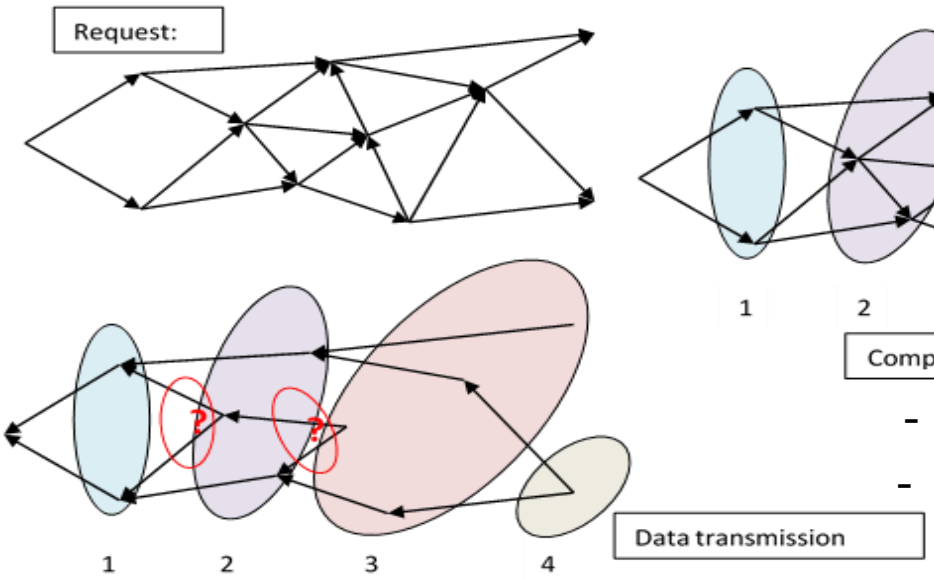
Flat: *Gradient Based Routing (GBR):*

- *Three phases:* (1) Request, (2) Computing the gradient, (3) Data transmission.
- *Typical application:* multiple sources → single destination (data collection).

(1) Request: the central node sends a request into the network by flooding.

(2) Computing the gradient: during the distribution of the request: "measurement" of the gradient → The gradient is the "shortest distance" from the central node: What is the lowest number of hops to reach the central node.

(3) Data transmission: selecting the shortest route and sending the data.

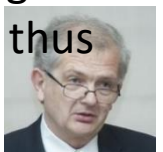


Data aggregation on the route is possible.

GBR variants:

In case of multiple equivalent route which one to select?

- *stochastic:* random selection
- *energy-based extension*: nodes having low energy can increase their "gradient", thus flow is oriented to another route.



Hierarchical: mentioning only two, descriptions can be found on the internet:

Low Energy Adaptive Clustering Hierarchy (LEACH): Hierarchical, based on dynamically formed clusters.

Geographic and Energy Aware Routing (GEAR): The nodes are familiar with their geographic location, and thus the messages are travelling only towards the targeted region.