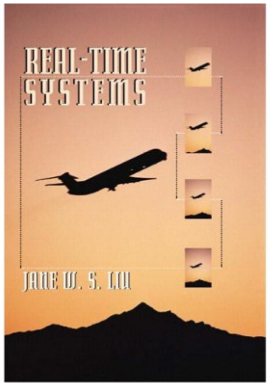
Real-Time Systems

Part 5: Real-time operating systems

Content

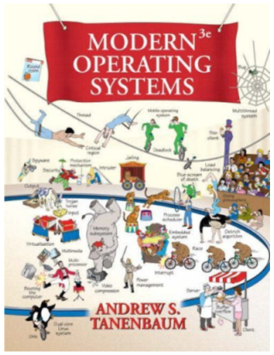
- Introduction
- Specific operating systems in detail:
  - Domain specific OSs:
    - OSEK
    - TinyOS
  - Classic real-time OSs
    - QNX
    - VxWorks
    - PikeOS
  - Linux- / Windows- real-time variants

Literature



Jane W. S. Liu, Real-Time Systems, 2000

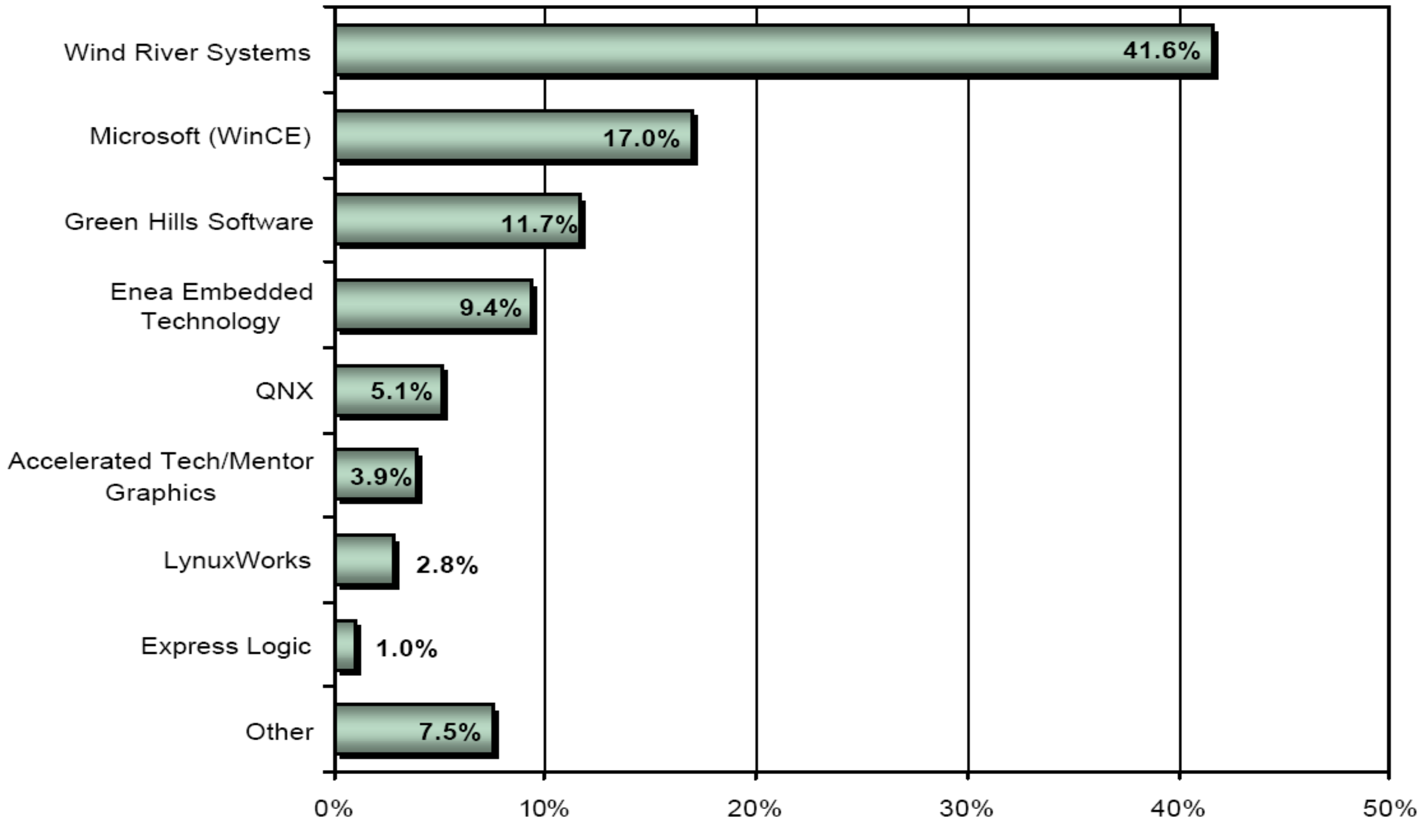
Dieter Zöbel, Wolfgang Albrecht: Echtzeitsysteme: Grundlagen und Techniken, 1995



Andrew S. Tanenbaum: Modern Operating Systems, 2008

Arnd Heursch et al.: Time-critical tasks in Linux 2.6, 2004
http://inf3-www.informatik.unibw-muenchen.de/research/linux/hannover/automation_conf04.pdf

Market share (2004)



Turnover market share, Overall turnover 493 Mio. Dollar, Source: The Embedded Software Strategic Market Intelligence Program 2005

Real-time operating system requirements

- Different and special requirements for real-time operating systems
  - Stable around-the-clock operation
  - Exact and predefined response times
  - Parallel processes
  - Support for multi-core and multi-cpu systems
  - Fast process-switching (lightweight process context information)
  - Real-time interrupt handling
  - Real-time scheduling
  - Real-time inter process communication
  - Extensive time services (absolute and relative clocks, alarm services)
  - Simple memory management

More real-time operating system requirements

- Support for I/O operations
  - Vast support for different peripherals
  - Direct user access to hardware-addresses and registers
  - Easy and fast driver development in user-space
  - Dynamic binding to the kernel
  - Direct usage of DMA
  - No intermediate buffering, user buffer directly to hardware
- Simple file systems
- Availability of protocols for field- or LAN-busses
- Modularized OS functionality with optional components (scalability)
- System API (e.g. POSIX)

Evaluation of real-time operating systems

- Important criteria for evaluation
  - Scheduling algorithm
  - Process management
  - Memory requirements (footprint)
  - Guaranteed response times

User- vs. Kernel-Space

- Normal processes run in user-space and make use of the functionality provided by the OS
- User-space has very often not the same access rights to memory/hardware as the OS, but profits from the OS services
- The kernel space has unlimited access to all resources and here the kernel runs and executes its services
- Interaction of the user-space with the kernel space happens via system calls

Scheduler

- Scheduling is the method by which processes (and threads) are given access to system resources (e.g. processor time)
- Schedulers will be treated in detail in one of the following lectures
- Important questions:
  - What kind of real-time scheduling algorithms/strategies are available?
  - Special concepts for periodic processes?
  - How is the problem of priority inversion treated?
  - When is the execution pre-emptible?

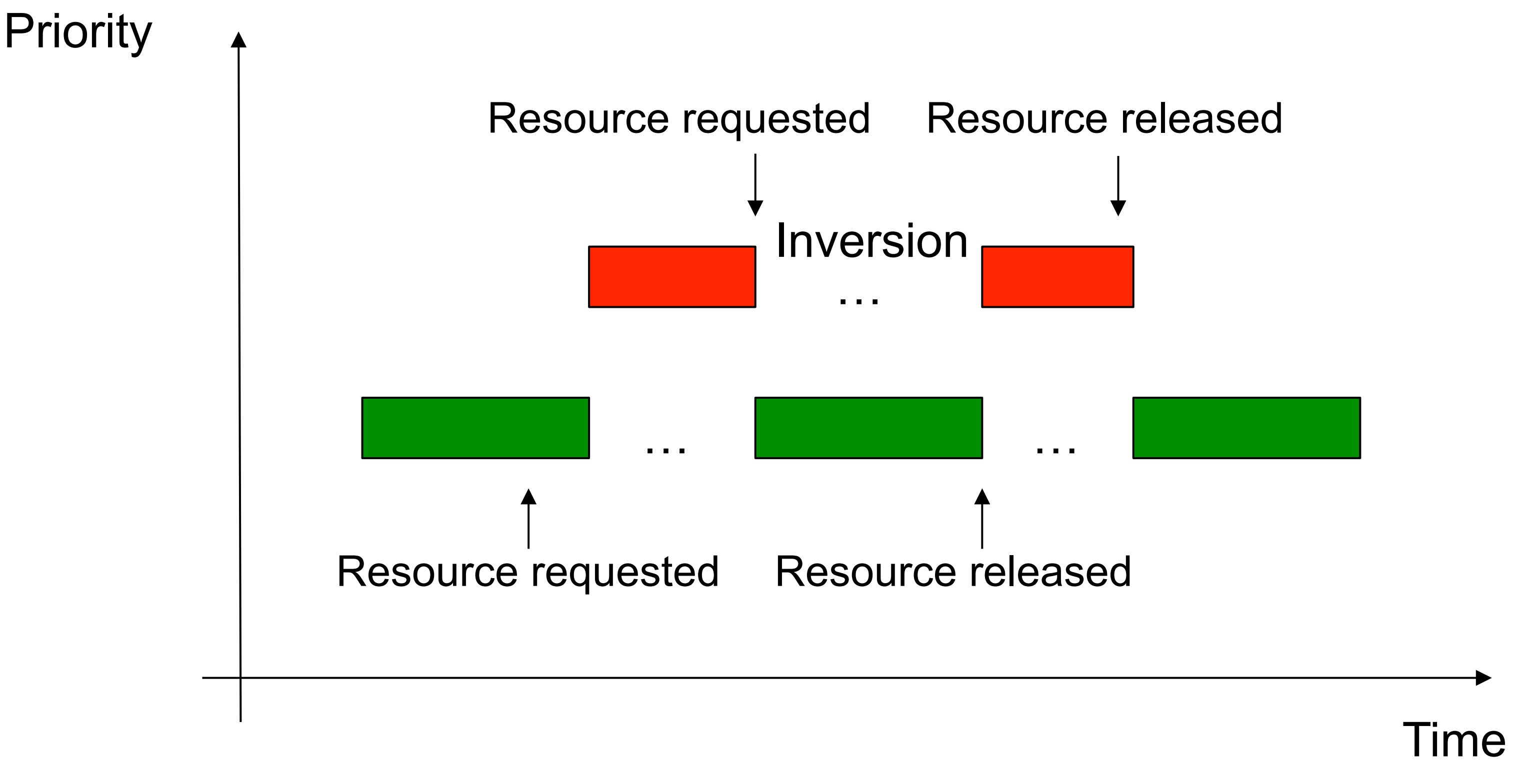
Scheduler

- Different classes of schedulers
  - **Cooperative or voluntary scheduler (non-preemptive)**: different processes can be run in parallel. But the processor can not be forcibly taken from a process, instead processes must cooperate with the scheduler (e.g. MacOS pre 9, Windows 3.1X and 16-bit processes Windows 95/98/ME)
  - **Preemptive scheduler**: the processor can be forcibly taken from a running process in users-pace (e.g. Linux, MacOS, Windows NT/XP/Vista/7)
  - **Preemptive OS**: the processor can be taken from any running process, even from processes running in kernel-space.
- Real-time systems are mostly preemptive

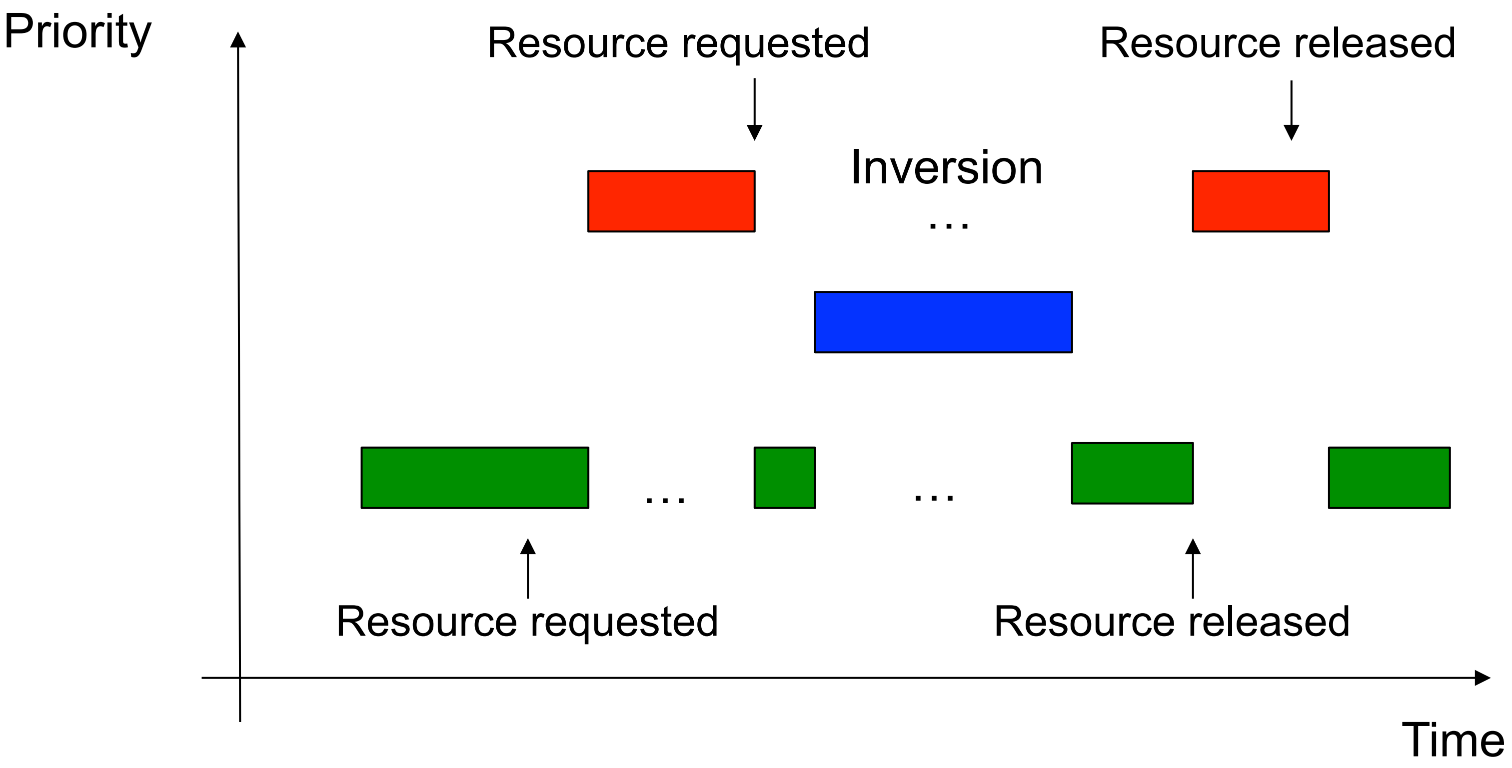
Priority inversion

- Processes have different priority but need a shared resource
- Lower priority has currently the resource, therefore the process with higher priority has to wait for the lower priority process
- If a third process with a priority between the priority of the two previous process exists and does not need the shared resource, then it will be scheduled since it is the process with highest priority and runnable. Therefore this medium priority process is preferred over the high priority process

Limited inversion



Unlimited inversion


Priority

Resource requested

Resource released

Inversion

…

…

…

Resource requested

Resource released

Time

Process management

- Evaluation of an OS according to:
  - Limited/Maximal number of processes
  - Methods for inter-process-communication (IPC)
  - Standard API compatibility (e.g. POSIX) for high portability

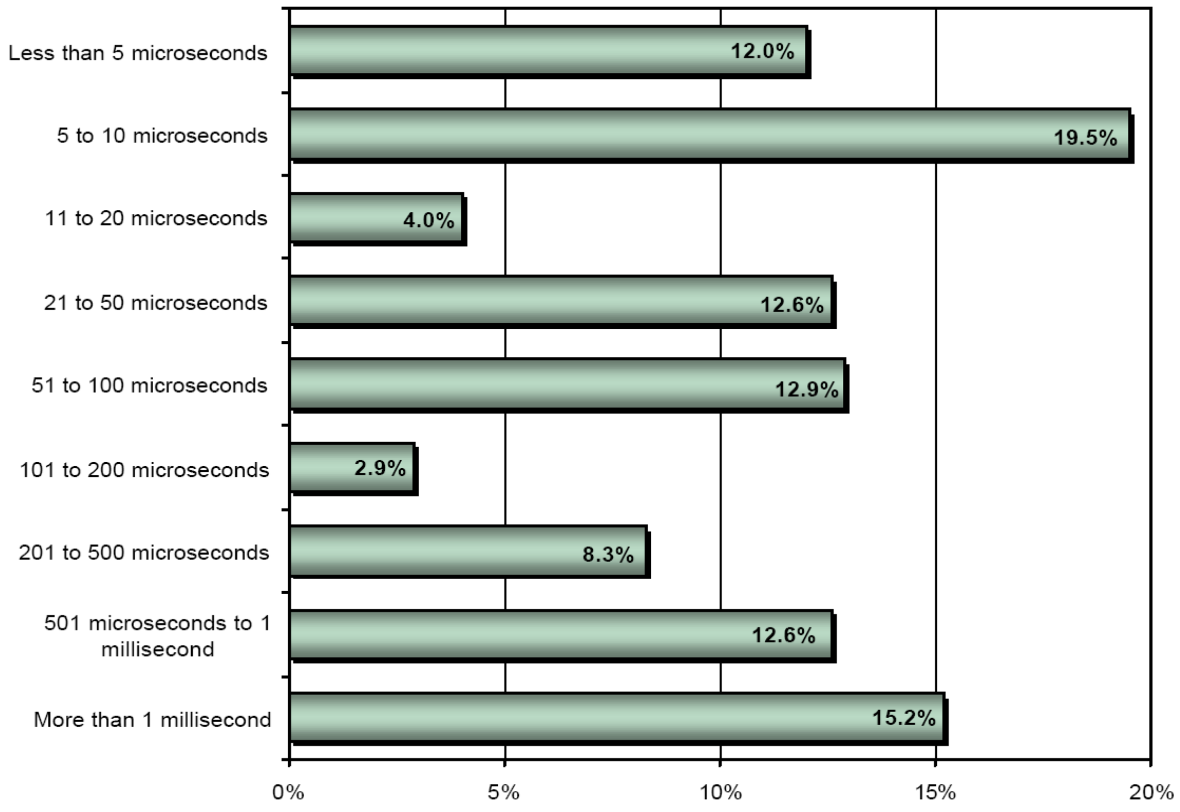
Memory consumption

- OS may needs to run on very different hardware platforms
  - Amount of available memory varies
  - Typical OS functionality and sub-systems very often not needed (e.g. file-systems, GUI)
- Real-time OS needs to be scalable
  - OS Modules must be selectable and optional according to the required functionalities
  - Minimal memory consumption/usage is crucial (**memory footprint**)

Response times

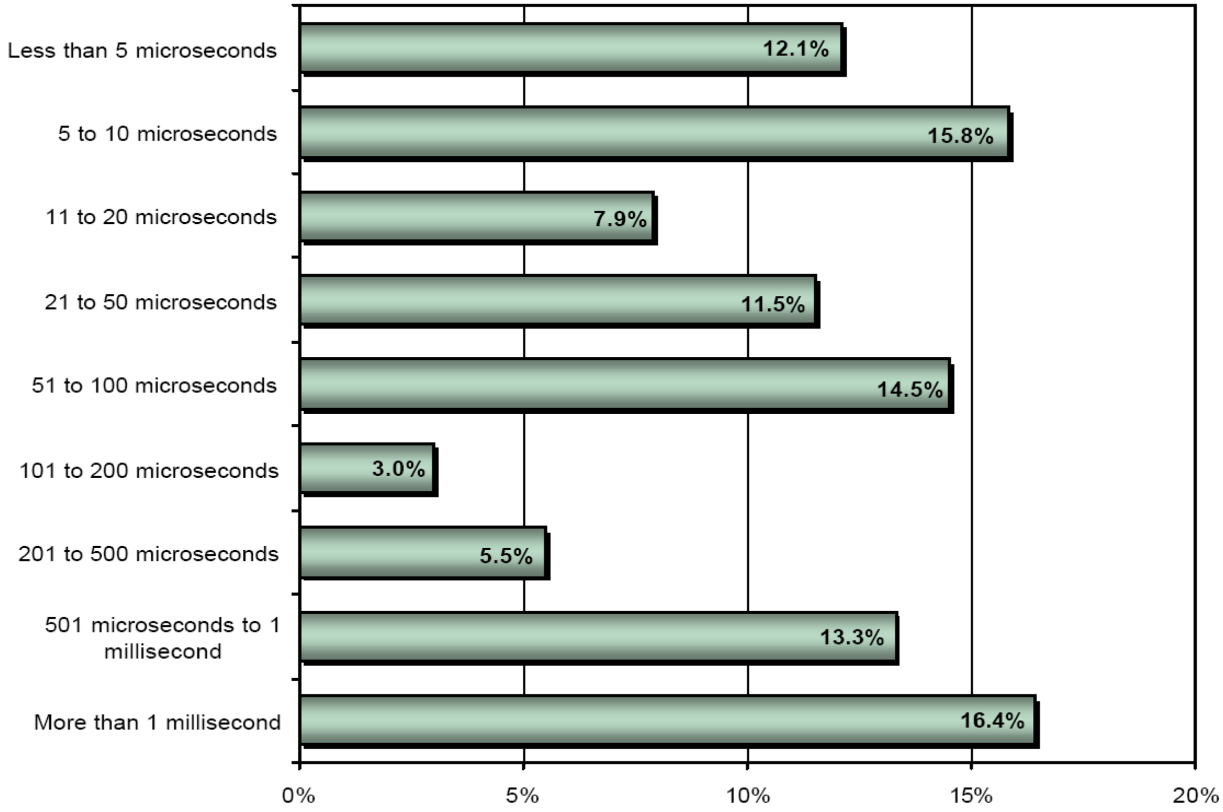
- Real-time capabilities defined by the following timing constraints:
  - **Interrupt latency**: amount of time needed between the occurrence of an interrupt and the execution of the first instruction of the corresponding ISR
  - **Scheduling latency**: amount of time between the execution of the last instruction of the ISR and the execution of the first instruction of the process, whose state changed to ready due to the interrupt
  - **Context switch latency**: elapsed time between the execution of the last instruction of a user-space process and the execution of the first instruction of the next process in user-space

Interrupt latency requirements



Typical response time requirements, Source: The Embedded Software Strategic Market Intelligence Program 2005

Context switch requirements



Typical context switching requirements, Source: The Embedded Software Strategic Market Intelligence Program 2005

Real-Time Operating Systems

OSEK/VDX

History and background

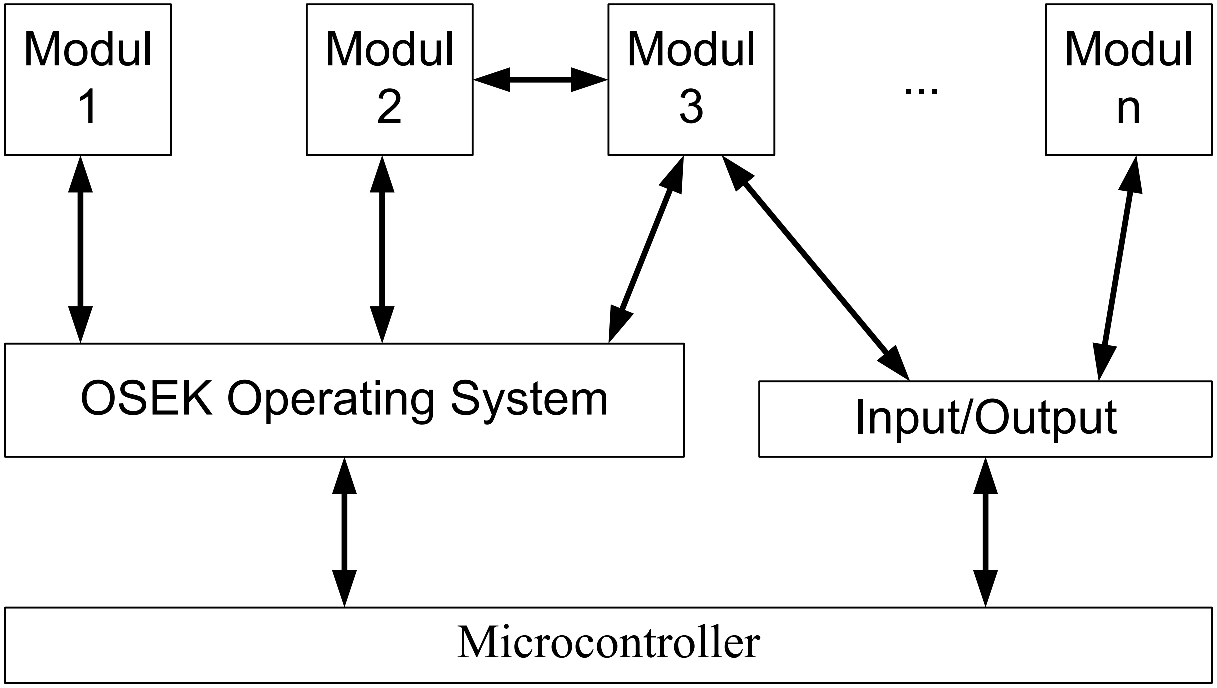
- Partner project of the German automotive industry in 1993 (BMW, Daimler, VW, Opel, Bosch, Siemens)
- OSEK: **O**ffene **S**ysteme und deren Schnittstellen für die **E**lektronik im **K**raftfahrzeug
- In 1994 merged with the 1988 founded French VDX-initiative (**V**ehicle **D**istributed E**x**ecutive) (PSA (Peugeot,Citroën), Renault) and renamed to OSEK/VDX
- Goal: definition of a standard-API for embedded real-time systems in the automotive industry
- Open-Standard (http://www.osek-vdx.org)
- Open-source implementations are available

Design condsiderations and goals

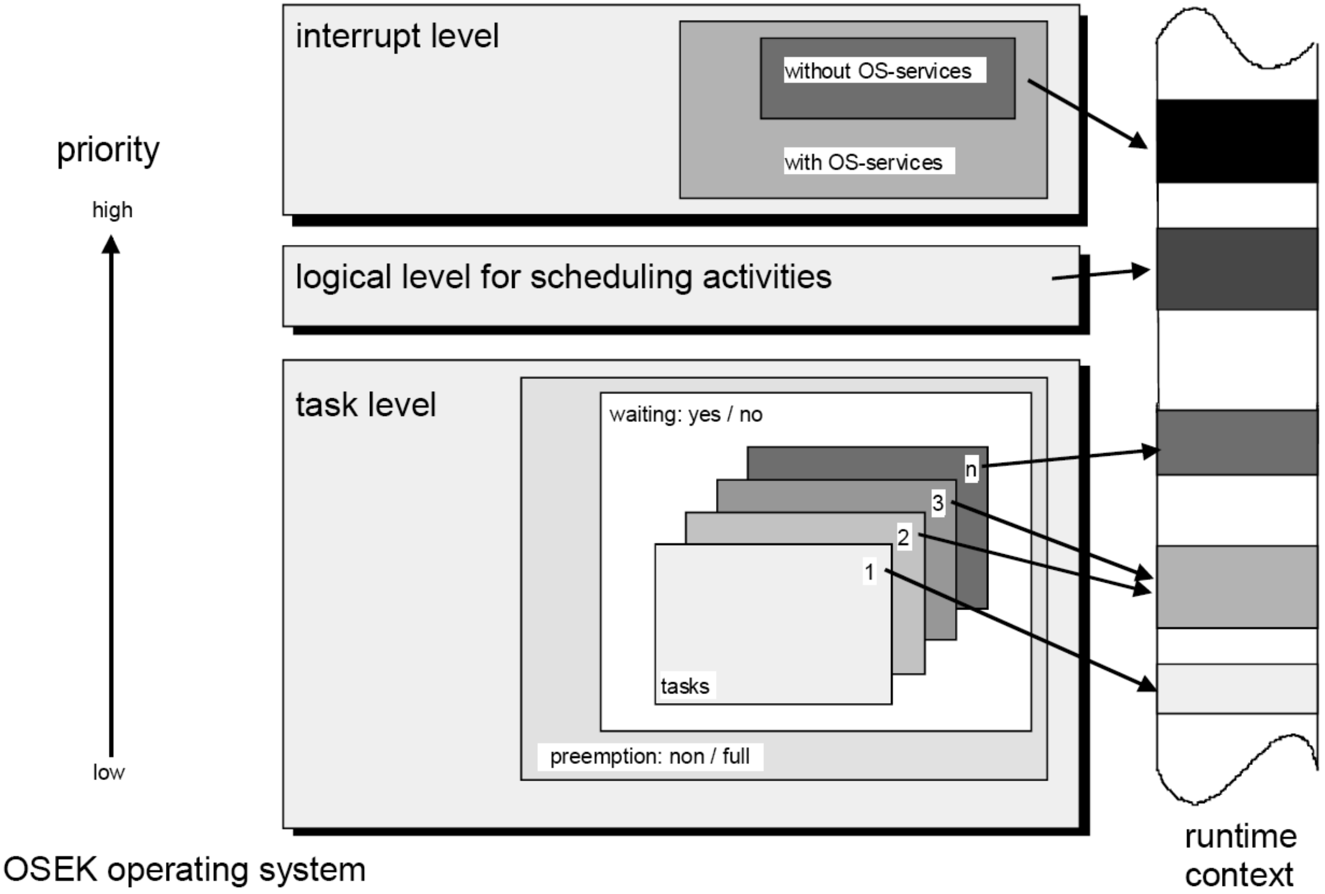
- Hard real-time constraints
- High security requirements for applications
- High performance requirements
- Distributed and heterogeneous hardware
- Typical real-time system requirements
- Other goals
  - Scalability
  - Simple configurability
  - Portability
  - Statically allocated OS

OSEK architecutre

- The interface between the single application-modules is standardized for high portability
- I/O operations are not exactly specified



Execution layers in OSEK



OSEK: scheduling and processes

- Scheduling: only scheduling with static priorities
- Processes:
  - OSEK specifies two kinds of processes:
    1. Basic process
    2. Extended process: has the ability to wait and handle external asynchronous events by using the waitEvent() system call
  - Processes can be programmed to be preemptive or non-preemptive
  - Process states in OSEK: running, ready, waiting, suspended.

OSEK: interrupt handling

- Two different categories:
  - ISR category 1: without OS-services
    - typically for fast and high-priority interrupts
    - After ISR execution the interrupted process is resumed
  - ISR Kategorie 2: with OS service routines
    - After ISR execution the scheduler chooses the next process to be run

OSEK: Priority inversion

- To avoid priority inversion and deadlocks OSEK uses an immediate priority ceiling protocol
  - Every resource has an upper priority bound (maximum priority of the processes using the resource)
  - If a process requests a resource then the process priority is lifted to the upper bound
  - On release the priority drops to the original priority

Real-Time Operating Systems

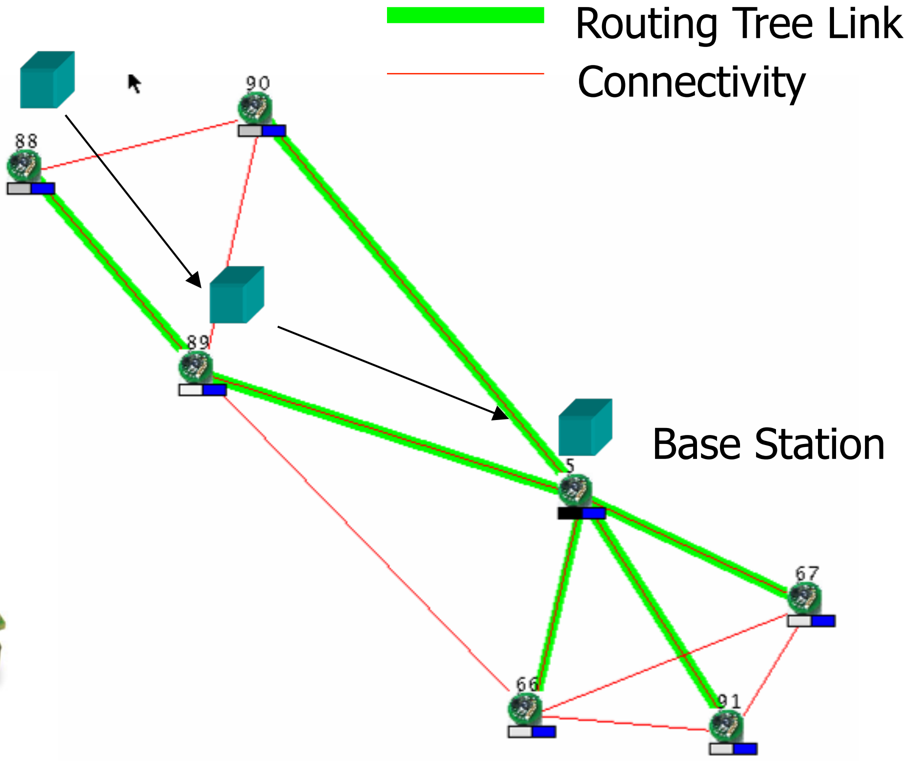
TinyOS

History and background

- TinyOS began as a project at UC Berkeley in co-operation with Intel Research and Crossbow Technology
- Written in the nesC (**n**etwork **e**mbedded **s**ystems C, C-dialect) programming language, a component-based and event-driven language
- TinyOS programs are built out of components, some of which present hardware abstractions and other packet communication, routing, sensing, actuation and storage
- Components are connected using interfaces
- Optimized for sensor networks
- Open-Source

Field of application: AdHoc sensor networks

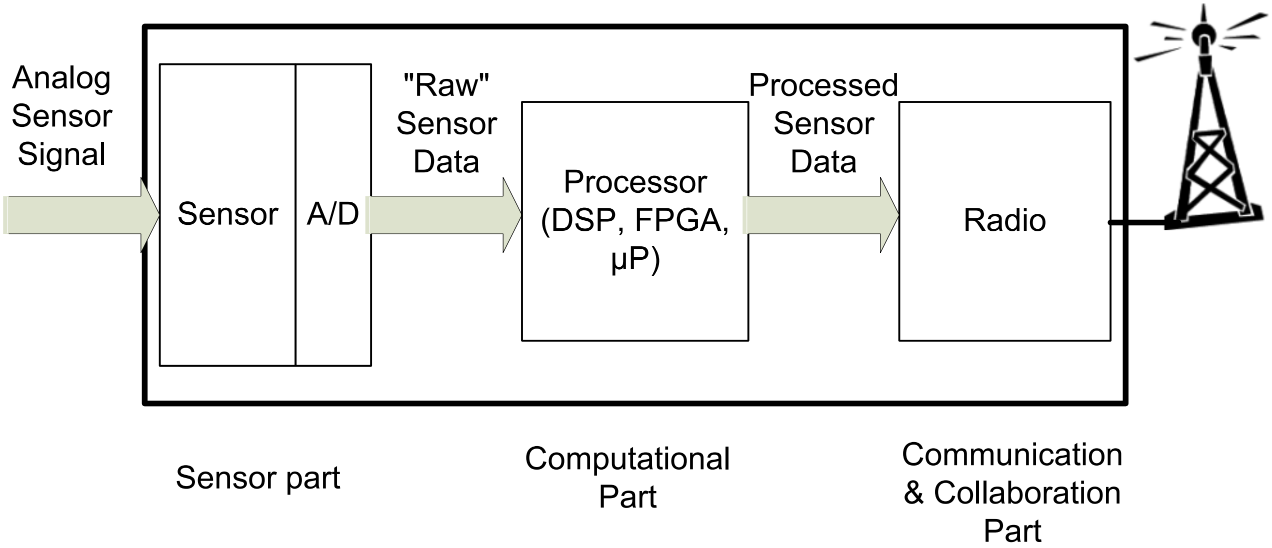
- Smart-Dust: many small sensors monitor the environment
- Goal: robust and large scale sensing
- Challenges:
  - Restricted life-time (battery)
  - Restricted memory
  - Small bandwidth
  - Small computational resources



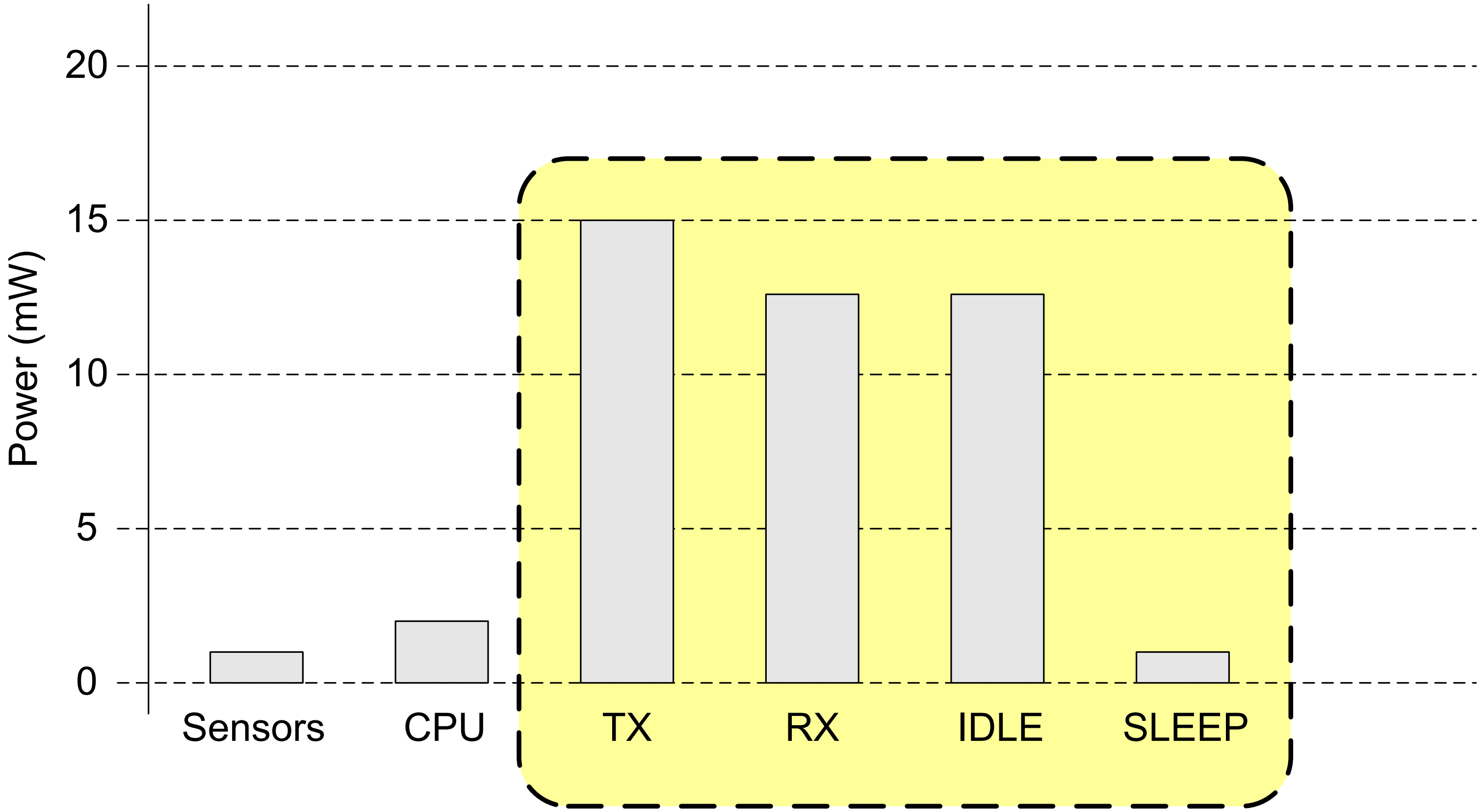
Quelle: http://tinyos.millennium.berkeley.edu

Hardware

- CPU: 4MHz, 8Bit, 512 Byte Ram
- Flash-memory: 128 kByte
- Radio-module: 2,4 GHz, 250 kbps
- Different sensor modules: e.g. A/DC, light, air-pressure

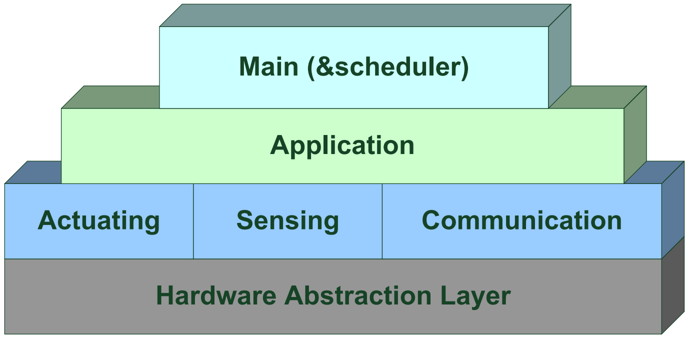


Power consumption

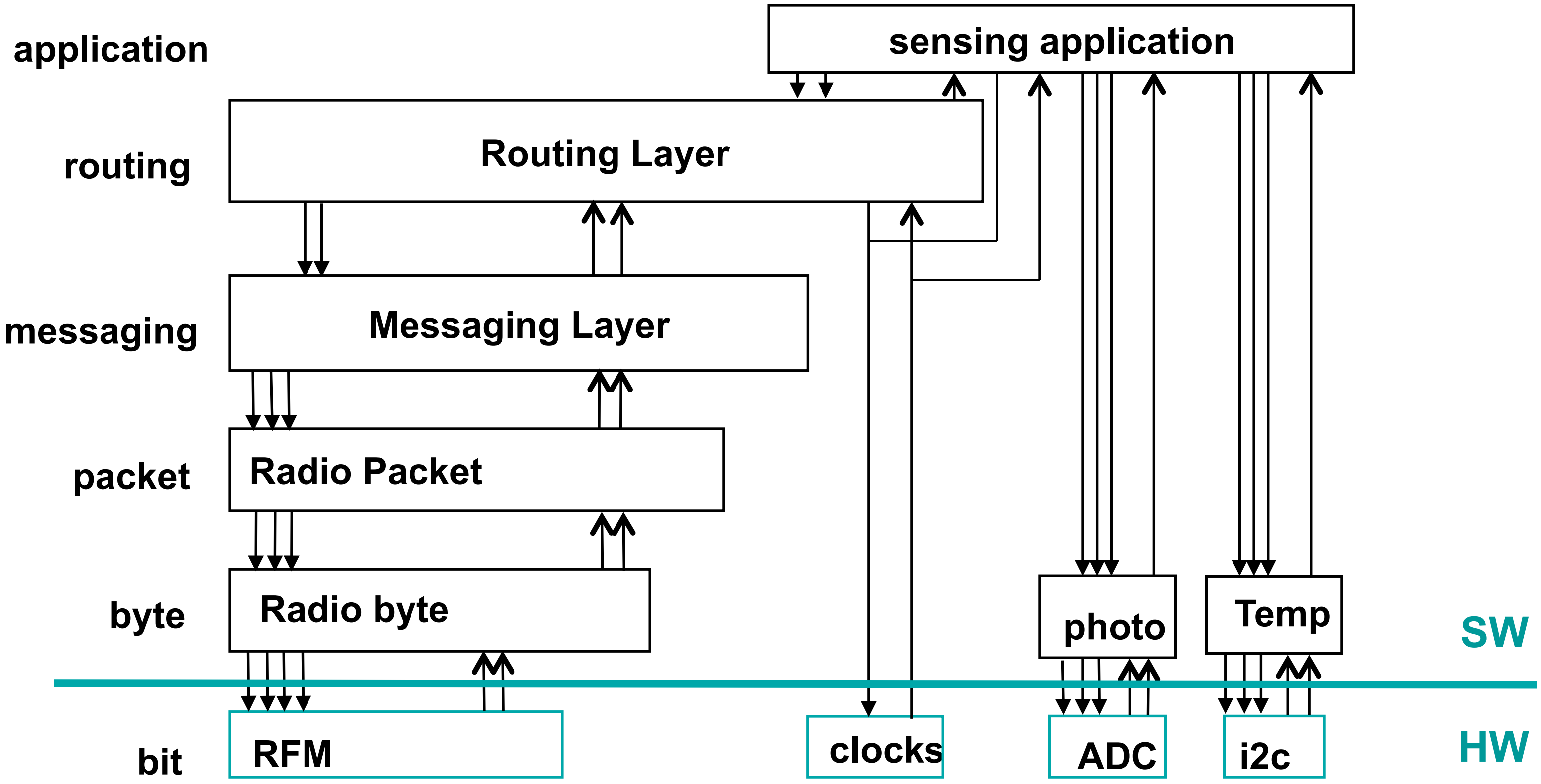


TinyOS

- TinyOS is not a real OS in the traditional sense, but an application-oriented OS
  - Single program, no separate OS and applications
  - No kernel, no processes and no memory-management
  - single shared stack
  - event-based execution model
  - Static memory allocation
- Concurrency model:
  - Execution in different contexts: foreground tasks for interrupt-events or background-tasks
  - Processes are only interrupted by events, but not other processes
  - Task scheduling: FIFO



TinyOS - architecture



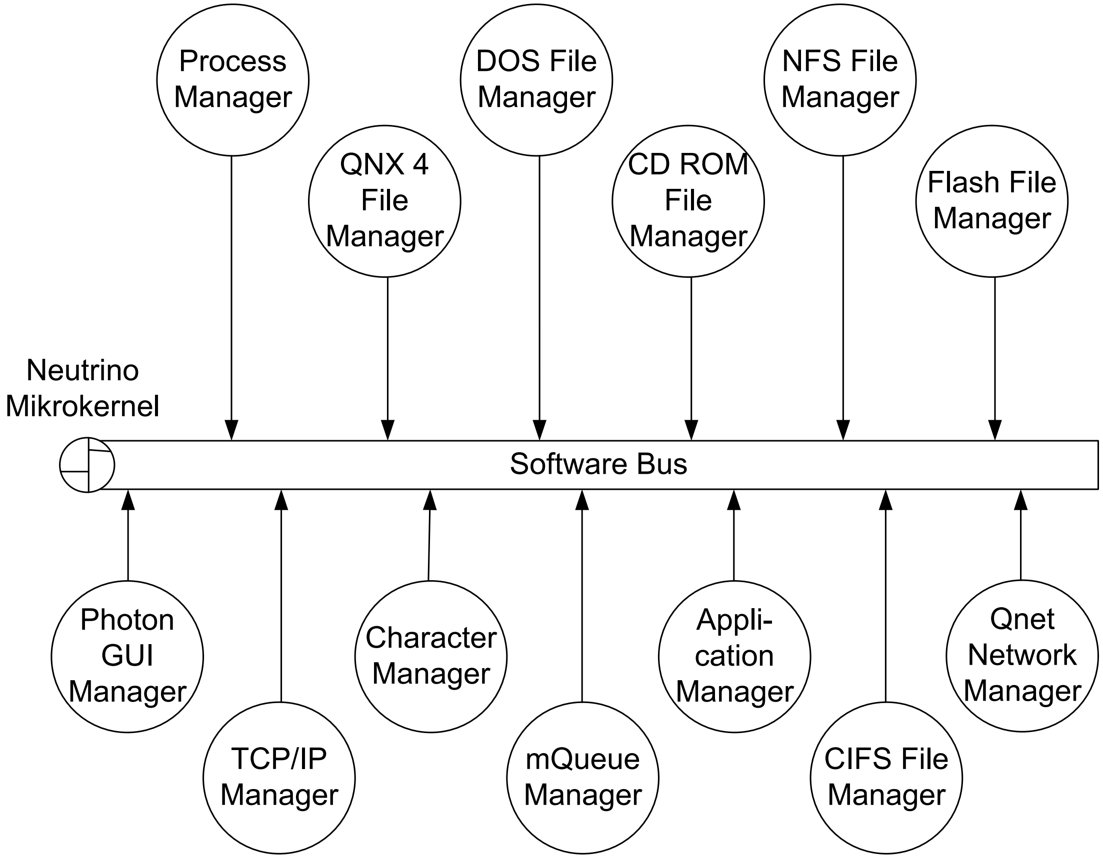
Real-Time Operating Systems

QNX

History and background

- 1980 Gordon Bell and Dan Dodge develop their own real-time operating system
- Toward the end of the 1990s a complete new version was developed: QNX Neutrino RTOS
- Extremely small and scalable micro-kernel (few kb) architecture based on message passing
- POSIX certified
- Certified for lots of special standards (medical, automotive …)
- QNX Software systems was bought by Harman International Industries and was later acquired by Research in Motion

QNX architecture



Real-time support

- Fast interrupt latencies and context switches for fast response time from embedded hardware
- Priority inheritance to eliminate priority inversion
- Simplified modeling of real-time activities through synchronous message passing
- Nested interrupts and a fixed upper bound on interrupt latency to ensure that high-priority interrupts are handled first, within a predictable timeframe
- High-availability and broad support for different usages and hardware scenarios

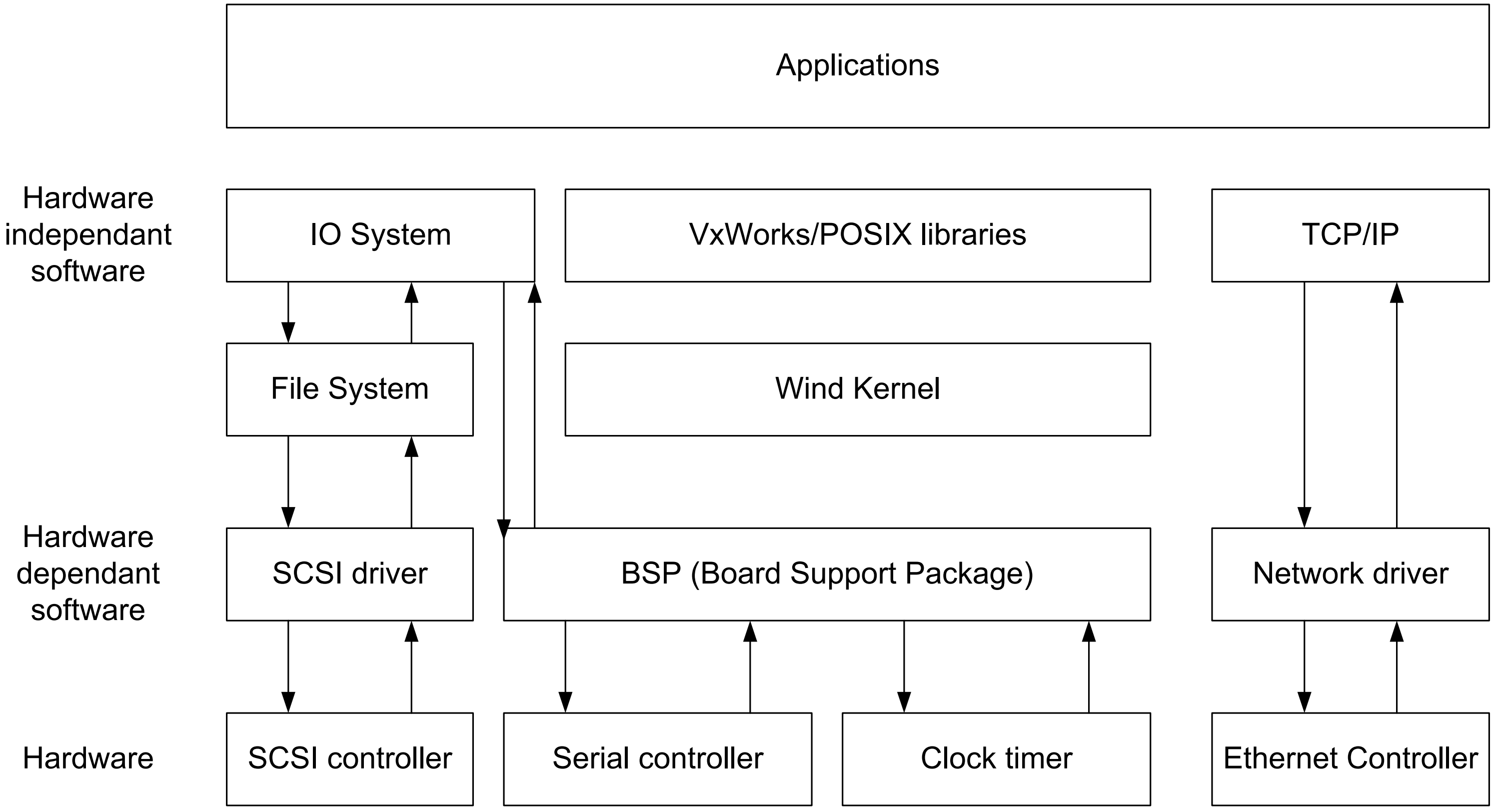
Real-Time Operating Systems

VxWorks

History and background

- VxWorks started as a set of enhancements to the VRTX kernel sold by Ready Systems
- Wind River replaced the kernel in 1987 with their own
- Wind River was acquired in 2009 by Intel
- POSIX certified
- Integrated development and deployment using Eclipse
- Currently one of the market leaders
- Wind River has also their own real-time embedded Linux distribution

VxWorks architecture


Applications

Hardware independant software

IO System

VxWorks/POSIX libraries

TCP/IP

File System

Wind Kernel

Hardware dependant software

SCSI driver

BSP (Board Support Package)

Network driver

Hardware

SCSI controller

Serial controller

Clock timer

Ethernet Controller

Real-Time Operating Systems

PikeOS

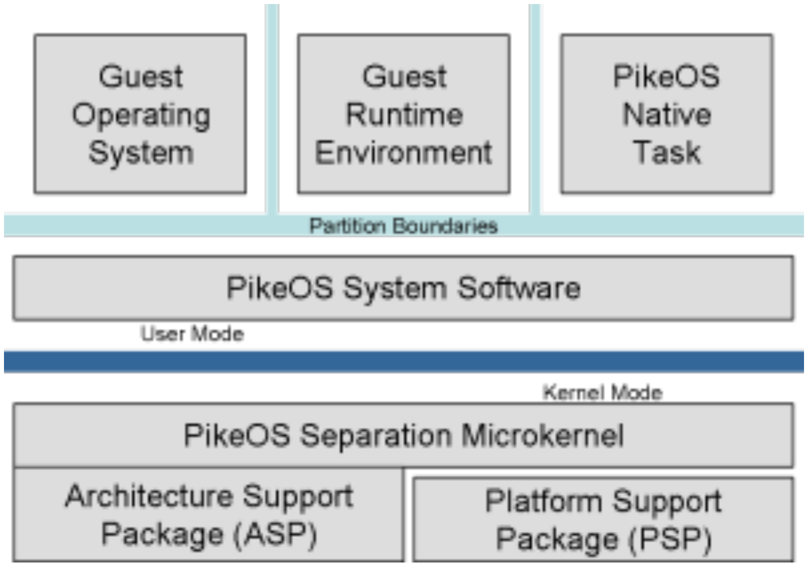
History and background

- **PikeOS** is a microkernel-based real-time operating system made by SYSGO AG
- Main idea: virtualization of the hardware - this allows for hard real-time systems to be virtualized, while still retaining their timing properties
- Guest OSs must be adapted
- POSIX certified (certified for lots of other standards)
- Open-source alternatives exist like L4

PikeOS: OS with para-virtualization

- Several guest OS can run in parallel on the same hardware
- Memory regions and CPU times of the partitions can be assigned statically during the implementation
- Advantages due to the partitioning:
  - For certification only the safety-critical parts need to be certified
  - Number of ICUs can be reduced drastically by merging them
  - Real-time components can be easily separated from non critical components – easier to proof fulfillment of deadlines

Architecture



Real-Time Operating Systems

Linux

History and background

- Well known OS initially created by Linus Torvalds
- Although widely used on servers, desktops and embedded systems – it is not out of the box usable for real-time systems
- POSIX support
- Easy development, portable and good hardware support
- Problems in the vanilla kernel
  - due to big kernel lock (removed in 2.6.37) and other critical sections
  - Insufficient timer resolution
  - Non-preemptive kernel

Linux: Real-time kernel configuration

- Real-time patches and configuration necessary
- Kernel is made fully preemptible
  - In-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes
  - Priority inheritance for in-kernel spinlocks and semaphores.
  - Converting interrupt handlers into preemptible kernel threads
  - Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to userspace POSIX timers with high resolution

Linux: Memory management

- Linux supports virtual memory
- Usage of virtual memory leads to non-deterministic latencies (e.g. page swapping)
- Solution: static memory allocation and locking of all pages (**pinning**) using the `mlock()` and `mlockall()` functions.
- **Pinning** makes memory pages resident, so that the locked pages are never swapped.

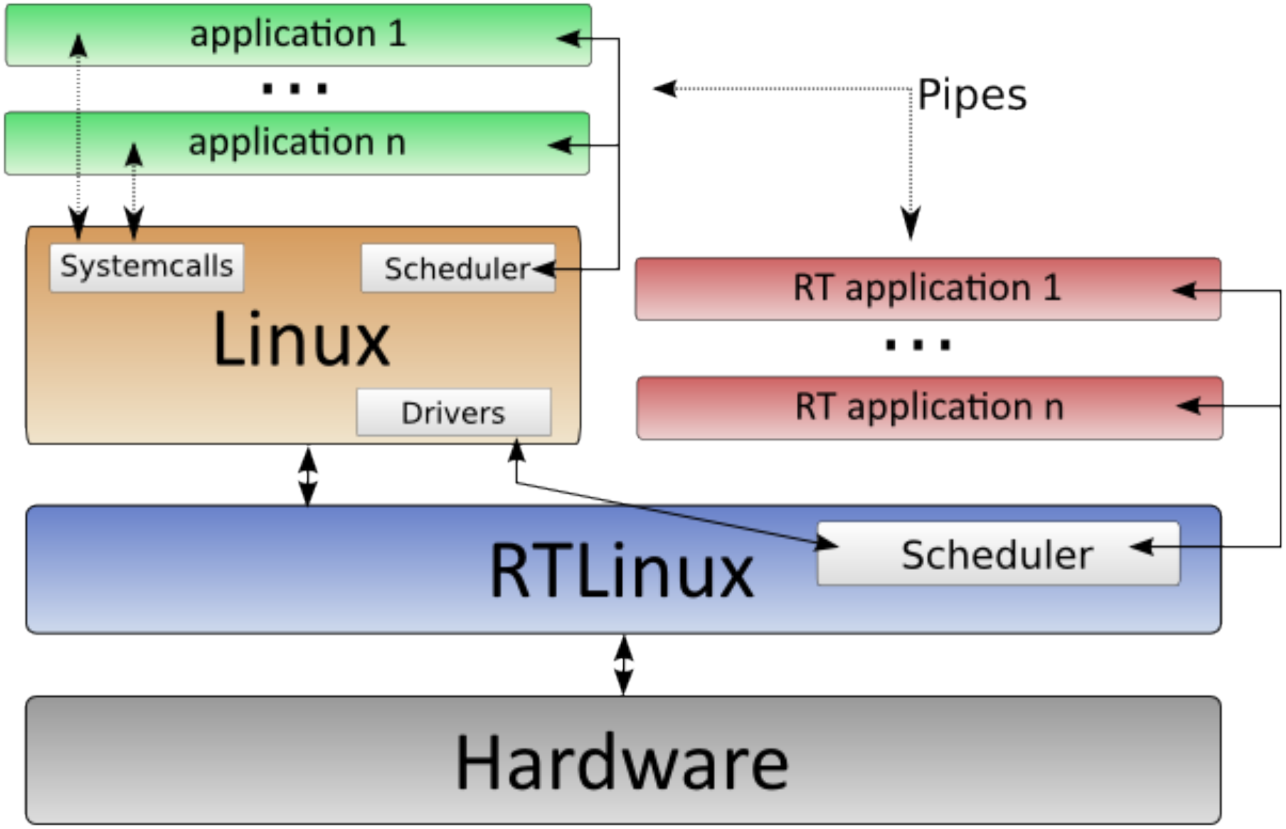
Real-Time Operating Systems

RTLinux/RTAI

RTAI / RTLinux approach

- RTAI/RTLinux add a new layer between the hardware and the kernel
    - Full control over the layer above interrupts
    - Virtualization of interrupts (Barabanov, Yodaiken, 1996): interrupts are converted to messages, which can be delivered to their designated targets
    - Virtualization of the clock
    - Virtual functions to enable/disable interrupts
    - The linux system runs as process with lowest priority
- RTAI additionally adds Hardware Abstraction Layer (HAL) between hardware and kernel (just a few dozen lines of code)
- Both make use of loadable kernel modules for real-time applications

RTLinux architecture



Real-Time Operating Systems

Windows Embedded

Windows embedded family

- Windows Embedded Compact
(previously named Windows Embedded CE and Windows CE)
  - Real-time operating system
  - Footprint can be as small as 1 MB
  - Modularized components
  - Supports ARM, MIPS and x86 processors
- Windows Embedded Standard
(previously named Windows XP Embedded)
  - Fully modularized Windows XP/ Windows 7
  - Full Win32 API, runs only on x86

Summary

- There is no definitive real-time OS – every field of application has their own demands and requirements
- The minimum memory consumption ranges from a few kilobytes to several megabytes
- The OSs are typical scalable – for changing the available services either the system has to be recompiled (e.g. VxWorks) or different processes have to be loaded (e.g. QNX)
- Real-time capabilities of standard-OSs can be added using special extensions (e.g. RTLinux/RTAI)
- Most scheduling algorithms and IPC mechanisms are similar to the ones proposed in the POSIX standards