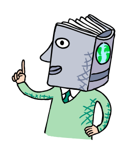
Embedded & Real-time Operating Systems

Peter Marwedel
TU Dortmund, Informatik 12
Germany

(2010年 11 月 24 日)
Subset of slides selected for EECS 222C.

Reuse of standard software components

Knowledge from previous designs to be made available in the form of **intellectual property** (IP, for SW & HW).



- Operating systems
- Middleware
- ….

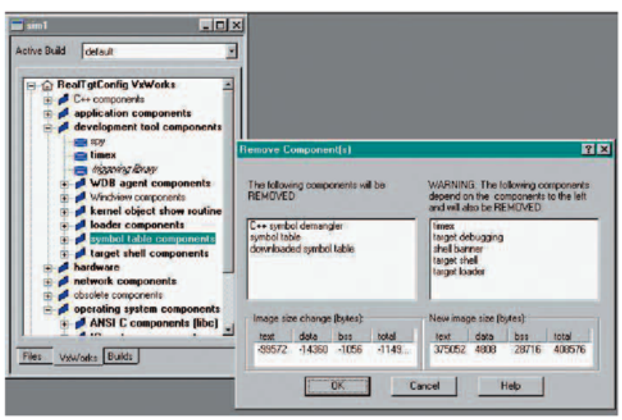
Embedded operating systems
- Characteristics: Configurability -

Configurability

No single OS will fit all needs, no overhead for unused functions tolerated ☞ configurability needed.

- Simplest form: remove unused functions (by linker ?).
- Conditional compilation (using #if and #ifdef commands).
- Dynamic data might be replaced by static data.
- Advanced compile-time evaluation useful.
- Object-orientation could lead to a derivation subclasses.

Example: Configuration of VxWorks

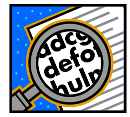


Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

http://www.windriver.com/products/development_tools/ide/tornado2/tornado_2_ds.pdf

© Windriver

Verification of derived OS?



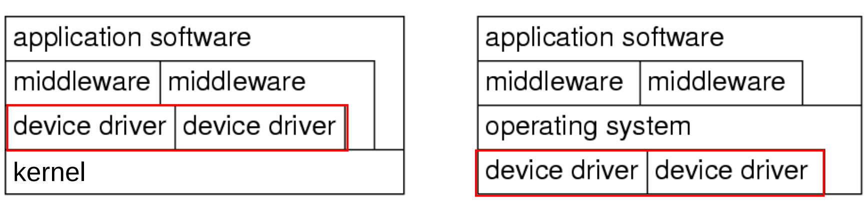
Verification a potential problem of systems with a large number of derived OSs:

- Each derived OS must be tested thoroughly;
- Potential problem for eCos
  (open source RTOS from Red Hat),
  including 100 to 200 configuration points
  [Takada, 01].

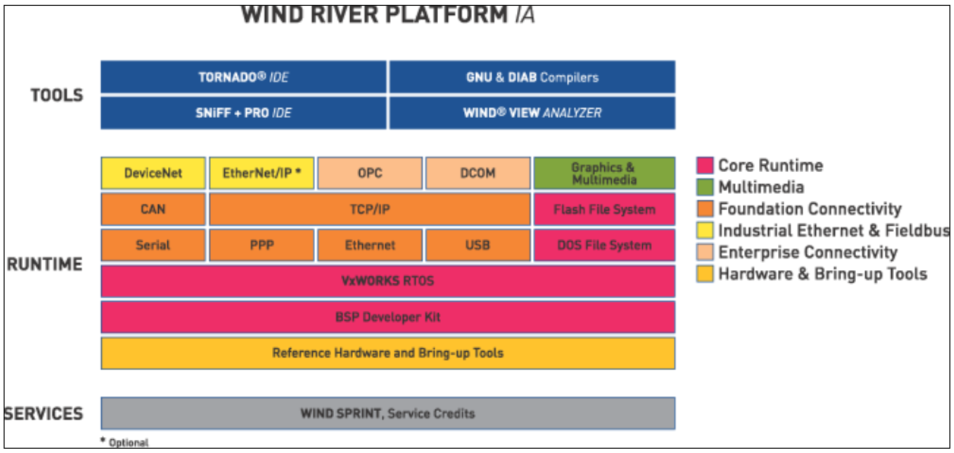
Embedded operating systems
- Disc and network handled by tasks -

- **Effectively no device that needs to be supported by all variants of the OS**, except maybe the system timer.
- Many ES without disc, a keyboard, a screen or a mouse.
- Disc & network handled by tasks instead of integrated drivers. Discs & networks can be handled by tasks.

Embedded OS | Standard OS

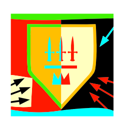


Example: WindRiver Platform Industrial Automation



Embedded operating systems
- Protection is optional-

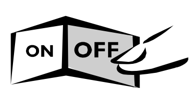
Protection mechanisms not always necessary:
ES typically designed for a single purpose,
untested programs rarely loaded, SW considered reliable.



Privileged I/O instructions not necessary and
tasks can do their own I/O.

Example: Let **switch** be the address of some switch
Simply use

```
load register,switch
```



instead of OS call.

However, protection mechanisms may be needed for safety and security reasons.

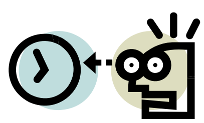
Embedded operating systems
- Interrupts not restricted to OS -

Interrupts can be employed by any process
For standard OS: serious source of unreliability.
Since

- embedded programs can be considered to be tested,
- since protection is not necessary and
- since efficient control over a variety of devices is required,
- it is possible to let interrupts directly start or stop tasks (by storing the task's start address in the interrupt table).
- More efficient than going through OS services.
- Reduced composability: if a task is connected to an interrupt, it may be difficult to add another task which also needs to be started by an event.

Embedded operating systems
- Real-time capability-

Many embedded systems are real-time (RT) systems and, hence, the OS used in these systems must be **real-time operating systems (RTOSs).**



Real-time operating systems
- Definition and requirement 1: predictability -

Def.: *(A) real-time operating system is an operating system that supports the construction of real-time systems.*

The following are the three key requirements

1. **The timing behavior of the OS must be predictable.**
   ∀ services of the OS: Upper bound on the execution time!
   RTOSs must be timing-predictable:
   - short times during which interrupts are disabled,
   - (for hard disks:) contiguous files to avoid unpredictable head movements.

[Takada, 2001]

Real-time operating systems
Requirement 2: Managing timing

2. **OS should manage the timing and scheduling**
   - OS possibly has to be aware of task deadlines; (unless scheduling is done off-line).
   - Frequently, the OS should provide precise time services with high resolution.

[Takada, 2001]

Real-time operating systems
Requirement 3: Speed

3. **The OS must be fast**

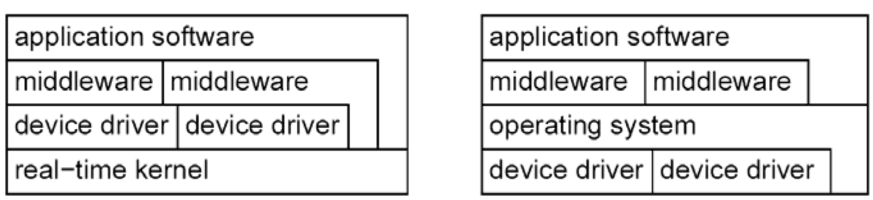
   Practically important.

[Takada, 2001]

RTOS-Kernels

Distinction between

- real-time kernels and modified kernels of standard OSes.



Distinction between

- general RTOSs and RTOSs for specific domains,
- standard APIs (e.g. POSIX RT-Extension of Unix, ITRON, OSEK) or proprietary APIs.

Functionality of RTOS-Kernels

Includes

- processor management,
- memory management,
- and timer management;

} resource management

- task management (resume, wait etc),
- inter-task communication and synchronization.

Classes of RTOSes according to R. Gupta: 1. Fast proprietary kernels

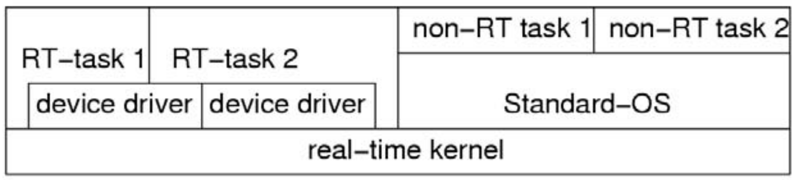
For complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect

[R. Gupta, UCI/UCSD]

Examples include
QNX, PDOS, VCOS, VTRX32, VxWORKS.

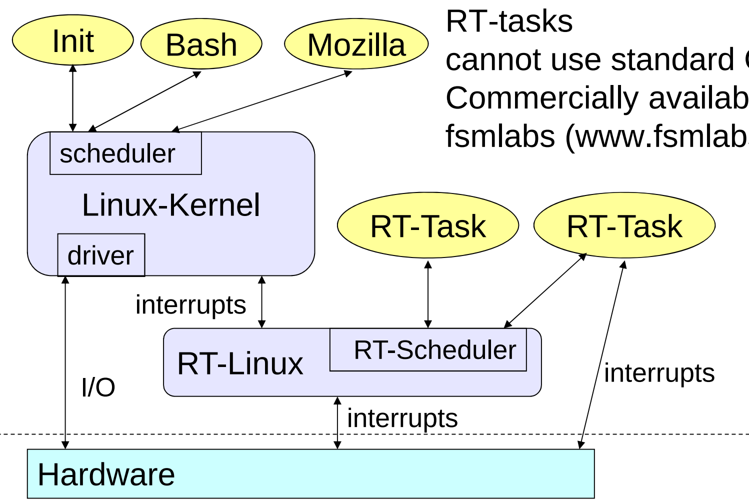
Classes of RTOSs according to R. Gupta: 2. RT extensions to std. OSs

Attempt to exploit comfortable main stream OS.
RT-kernel running all RT-tasks.
Standard-OS executed as one task.



+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services; less comfortable than expected

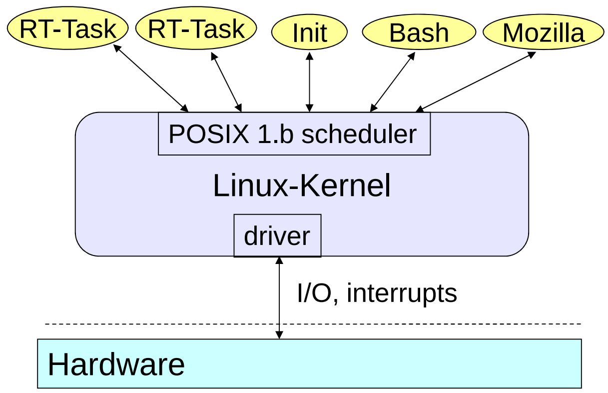
Example: RT-Linux



RT-tasks cannot use standard OS calls.
Commercially available from fsmlabs (www.fsmlabs.com)

Example:
Posix 1.b RT-extensions to Linux

Standard scheduler can be replaced by POSIX scheduler implementing priorities for RT tasks



Special RT-calls and standard OS calls available.
Easy programming, no guarantee for meeting deadline

Evaluation (Gupta)

According to Gupta, trying to use a version of a standard OS:
not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation.
Dependences between tasks not frequent for most applications of std. OSs & therefore frequently ignored.
Situation different for ES since dependences between tasks are quite common.

Classes of RTOSs (R. Gupta): 3. Research trying to avoid limitations

Research systems trying to avoid limitations. Include MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody

Research issues [Takada, 2001]:

- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- support for continuous media
- quality of service (QoS) control.