Introduction to Embedded Systems

5. Operating Systems

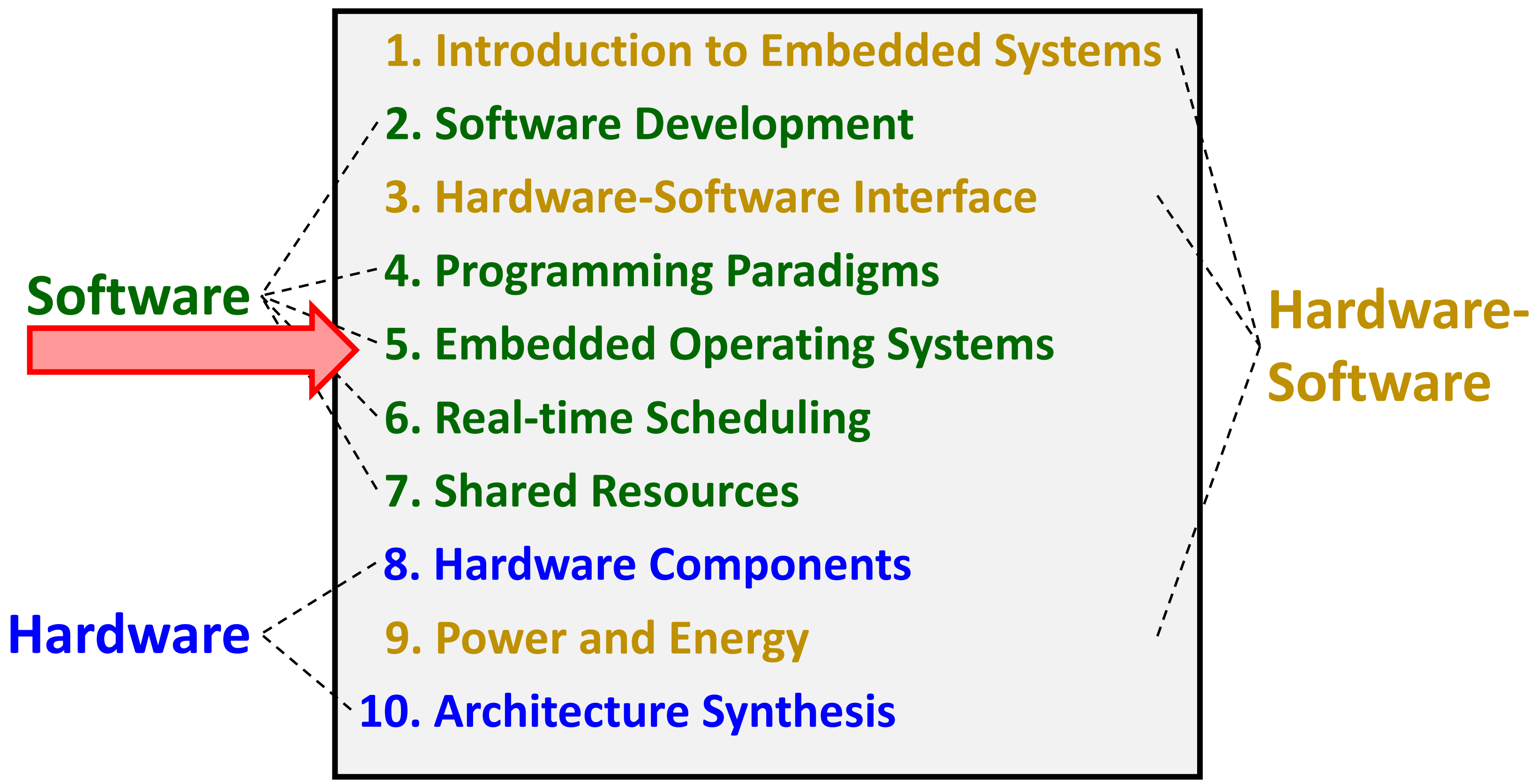
Prof. Dr. Marco Zimmerling

NETWORKED EMBEDDED SYSTEMS LAB

UNI FREIBURG

Embedded Operating Systems

Where we are …



1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis

Embedded Operating System: Motivation

- *Why an operating system (OS) at all?*
  - Same reasons why we need one for a traditional computer.
  - Not every device needs all services.

- In embedded systems we find a *large variety of requirements and environments:*
  - Critical applications with broad functionality (medical applications, space shuttle, process automation, …).
  - Critical applications with little functionality (ABS, pace maker, …).
  - Not very critical applications with broad range of functionality (smart phone, …).

Embedded Operating System: Motivation

- *Why is a desktop OS not suited?*
  - Monolithic kernel of a desktop OS often not modular, fault-tolerant, and configurable.
  - Typically offers many features that may not be needed and consume too many resources (e.g., energy, memory, compute time) for an embedded system.
  - Generally not designed for mission- or safety-critical applications. For example, the timing uncertainty may be too high to give any real-time guarantees.

Embedded Operating Systems

Essential characteristics of an embedded OS: *Configurability*

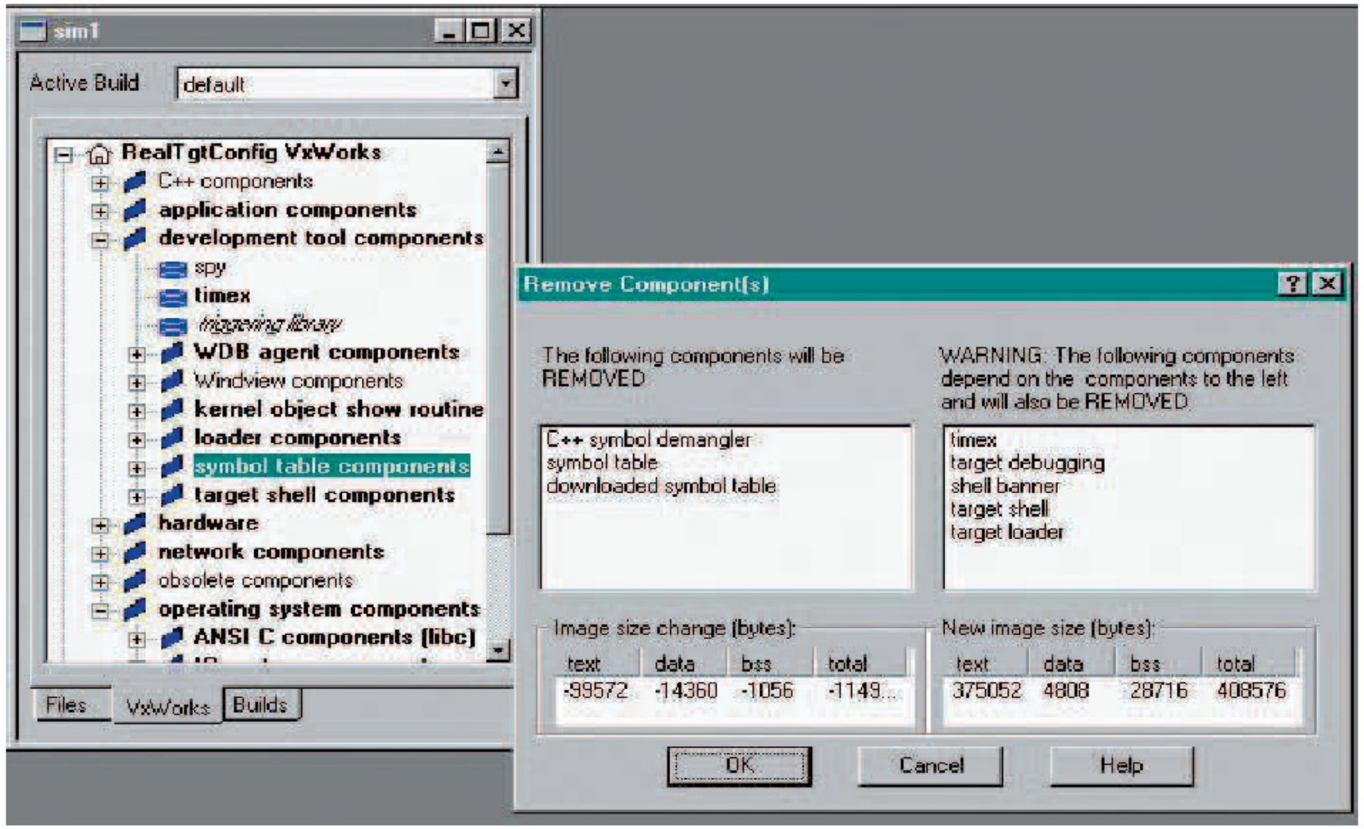
- *No single operating system will fit all needs*, but often no overhead for unused functions/data is tolerated. Therefore, configurability is needed.
- For example, there are many embedded systems without external memory, a keyboard, a screen, or a mouse.

Configurability examples*:*

- *Remove unused functions*/libraries (e.g., by the linker).
- *Use conditional compilation* (e.g., using `#if` and `#ifdef` commands in C).
- But deriving a *consistent configuration* becomes challenging if the set of possible combinations of functions is large (e.g., relevant components may be missed).

Example: Configuration of VxWorks

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



Active Build: default

- RealTgtConfig VxWorks
  - C++ components
  - application components
  - development tool components
    - spy
    - timex
    - *triggering library*
    - WDB agent components
    - Windview components
    - kernel object show routine
    - loader components
    - symbol table components
    - target shell components
  - hardware
  - network components
  - obsolete components
  - operating system components
    - ANSI C components (libc)

Remove Component(s)

The following components will be REMOVED

C++ symbol demangler
symbol table
downloaded symbol table

WARNING: The following components depend on the components to the left and will also be REMOVED

timex
target debugging
shell banner
target shell
target loader

Image size change (bytes):

text	data	bss	total
-99572	-14360	-1056	-1149...

New image size (bytes):

text	data	bss	total
375052	4808	28716	408576

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

© Windriver

Mars science rover

ASIMO robot

Real-Time Operating Systems (RTOS)

A real-time operating system is an operating system that supports the construction of real-time systems.

Key requirements:

1. *The timing behavior of the OS must be predictable.*

   For all services of the OS, an upper bound on the execution time is necessary. For example, for every service upper bounds on blocking times need to be available, i.e. for times during which interrupts are disabled. Moreover, almost all processor activities should be controlled by a real-time scheduler.

2. *The OS must manage timing and scheduling.*

   - OS has to be aware of deadlines and should have mechanism to take them into account in the scheduling.
   - OS must provide precise time services with a high resolution.

Embedded Operating Systems

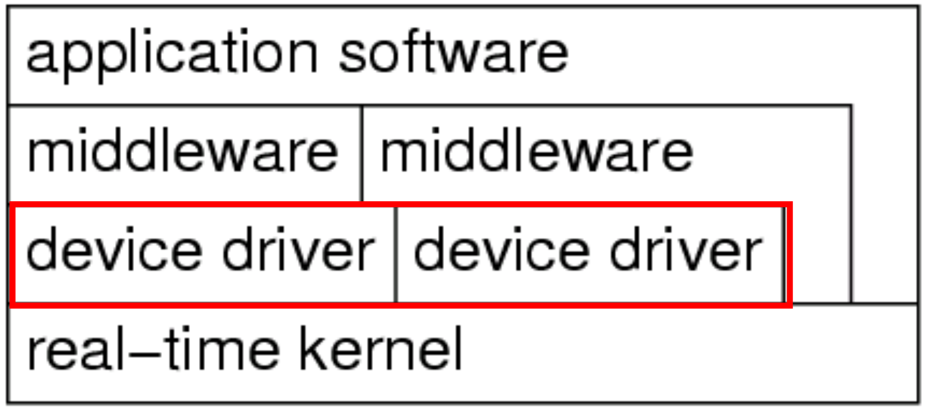
Features and Architecture

Embedded Operating System: Design Choices

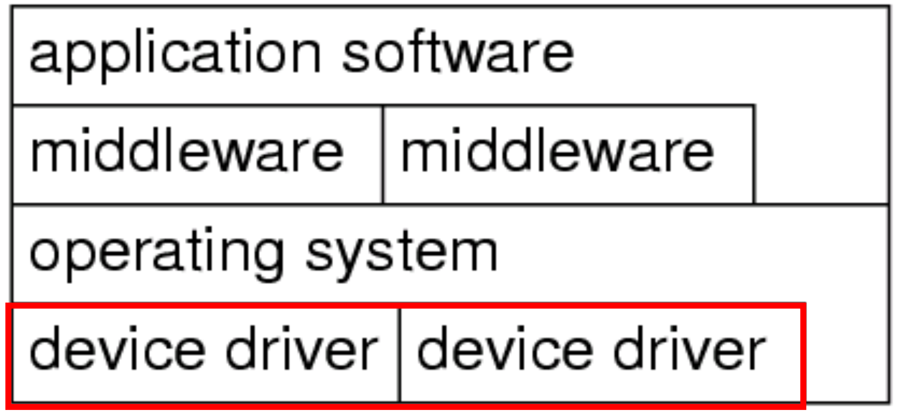
In an embedded OS, device drivers are typically handled directly by tasks rather than managed by the operating system itself:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler that runs within the real-time kernel.
- If several tasks use the same external device and the associated driver, then the access must be carefully managed (shared critical resource, ensure fairness of access).

Embedded OS



Standard OS



Embedded Operating Systems: Design Choices

Every task can perform an interrupt:

- In a *standard OS*, this would be a *serious source of unreliability*. But embedded programs are typically programmed in a controlled environment.
- It is possible to let *interrupts directly start or stop tasks* (by storing the start address of a task in the interrupt table). This approach is more efficient and predictable than going through the operating system's interfaces and services.

Protection mechanisms are not always necessary in embedded operating systems:

- Embedded systems are typically designed for a single purpose, untested programs are rarely loaded, and software can be considered to be reliable.
- However, protection mechanisms may be needed for *safety and security* reasons.

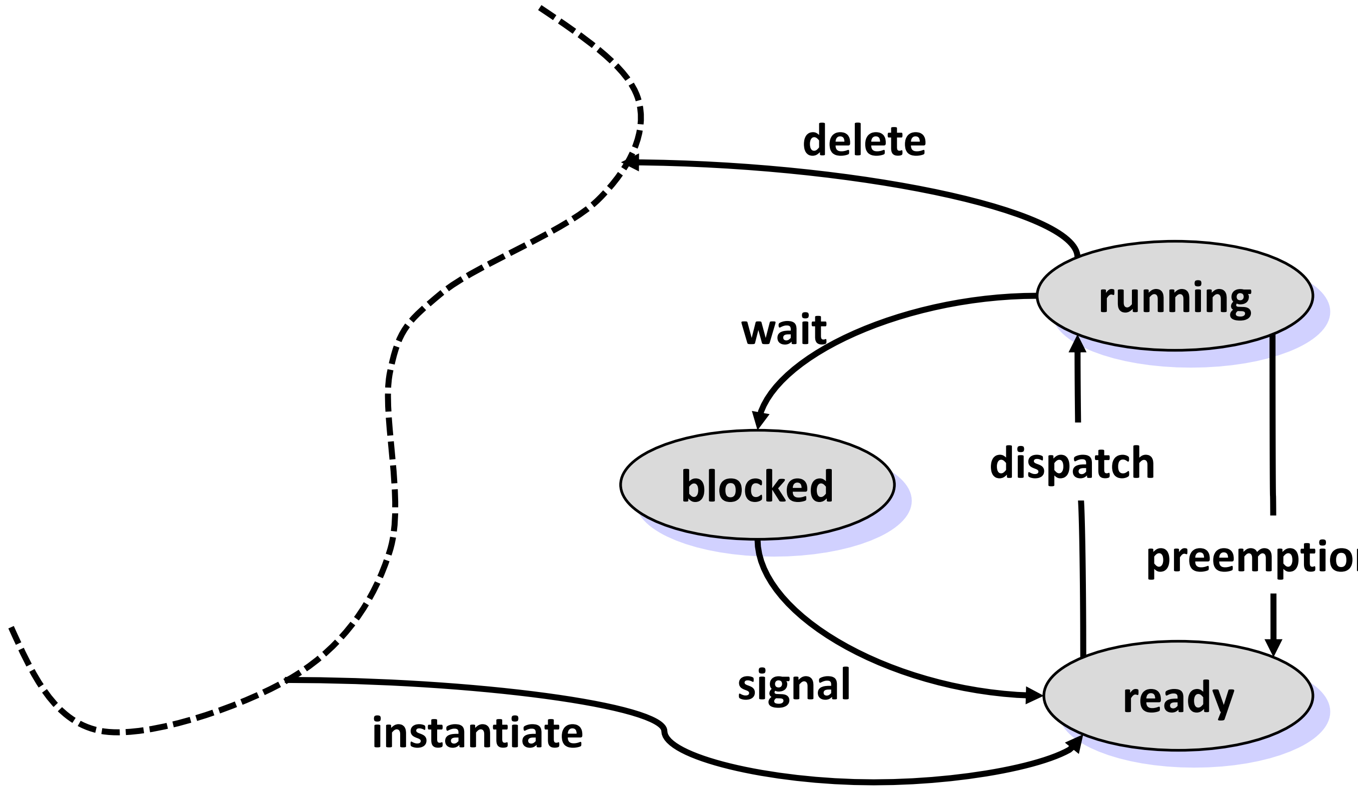
Main Functionality of Real-Time Operating System Kernels

Task management:

- *Execution of quasi-parallel tasks* on a processor using processes or threads (lightweight process) by
  - maintaining process states, process queuing,
  - allowing for preemptive tasks (fast context switching) and quick interrupt handling
- *CPU scheduling* (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- *Inter-task communication* (buffering)
- *Support of real-time clocks*
- *Task synchronization* (critical sections, semaphores, monitors, mutual exclusion)
  - In classical operating systems, synchronization and mutual exclusion are performed via semaphores and monitors.
  - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (e.g., priority inheritance protocols as described in a later chapter).

Task States

Minimal Set of Task States:



Task States

Running:

- A task enters this state when it starts executing on the processor. There is at most one task with this state in the system.

Ready:

- State of those tasks that are ready to execute but cannot be run because the processor is assigned to another task (i.e., another task is in state “running”).

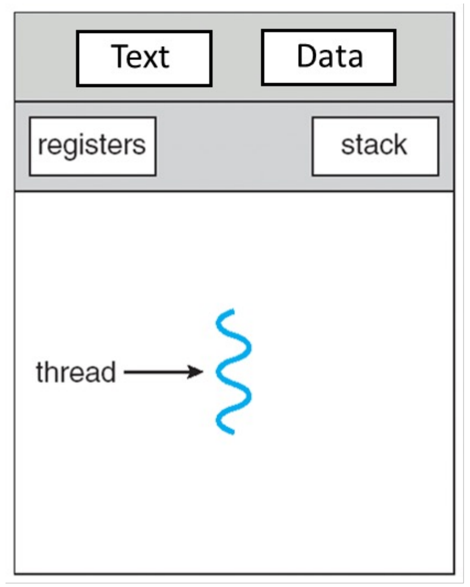
Blocked:

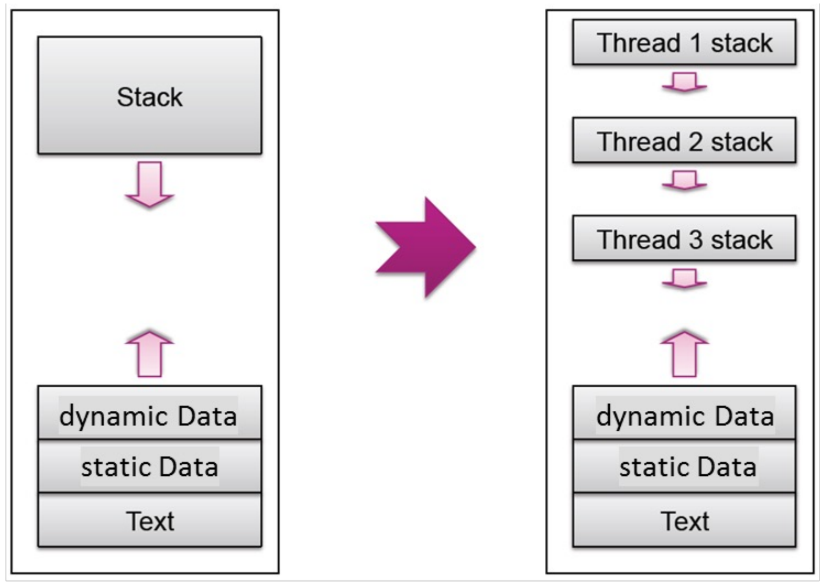
- A task enters this state when it executes a synchronization primitive to wait for an event (e.g., timer expires, mutually exclusive resource becomes available, data in a queue is read, or queue has again enough space to be written). In this case, the task is inserted in a queue associated with the event. The task at the head of the queue is resumed upon the occurrence of the corresponding event.

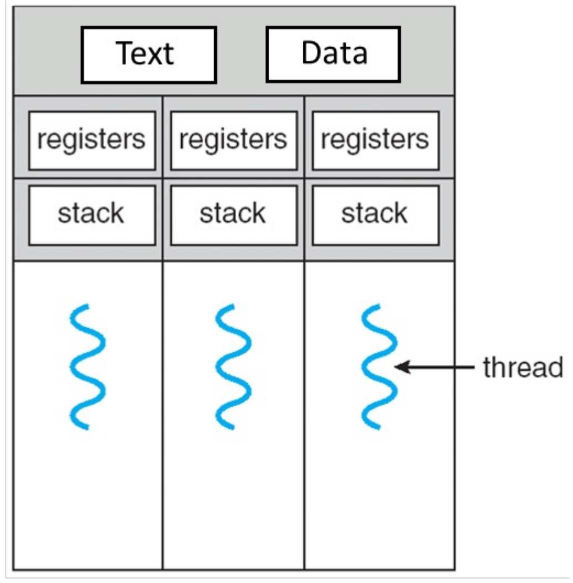
Threads

A thread is the smallest sequence of program instructions that can be managed independently by a scheduler. Thus, a thread is a basic unit of CPU utilization.

- *Multiple threads can exist within the same process* and share resources such as memory, while different processes do not share these resources:
  - Typically shared across different threads: memory.
  - Typically owned by each individual thread: registers and stack.







Threads

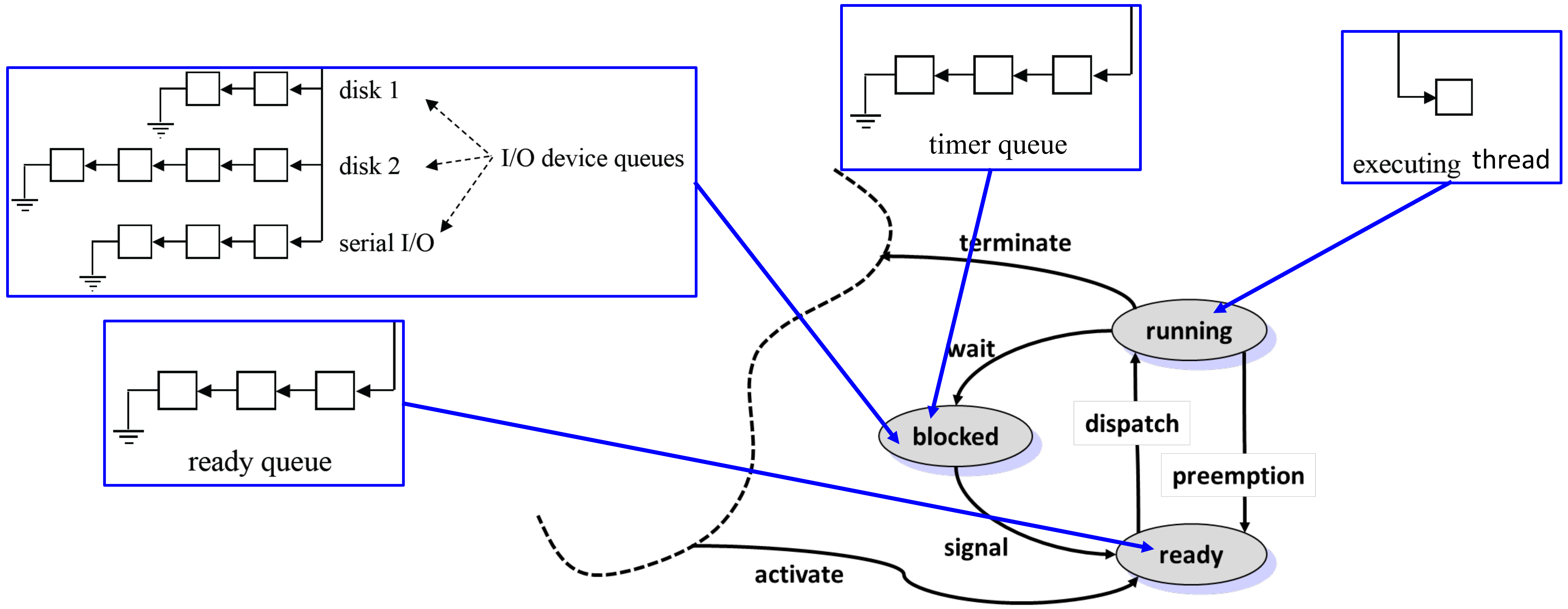
A thread is the smallest sequence of program instructions that can be managed independently by a scheduler. Thus, a thread is a basic unit of CPU utilization.

- *Multiple threads can exist within the same process* and share resources such as memory, while different processes do not share these resources:
  - Typically shared across different threads: memory.
  - Typically owned by each individual thread: registers and stack.

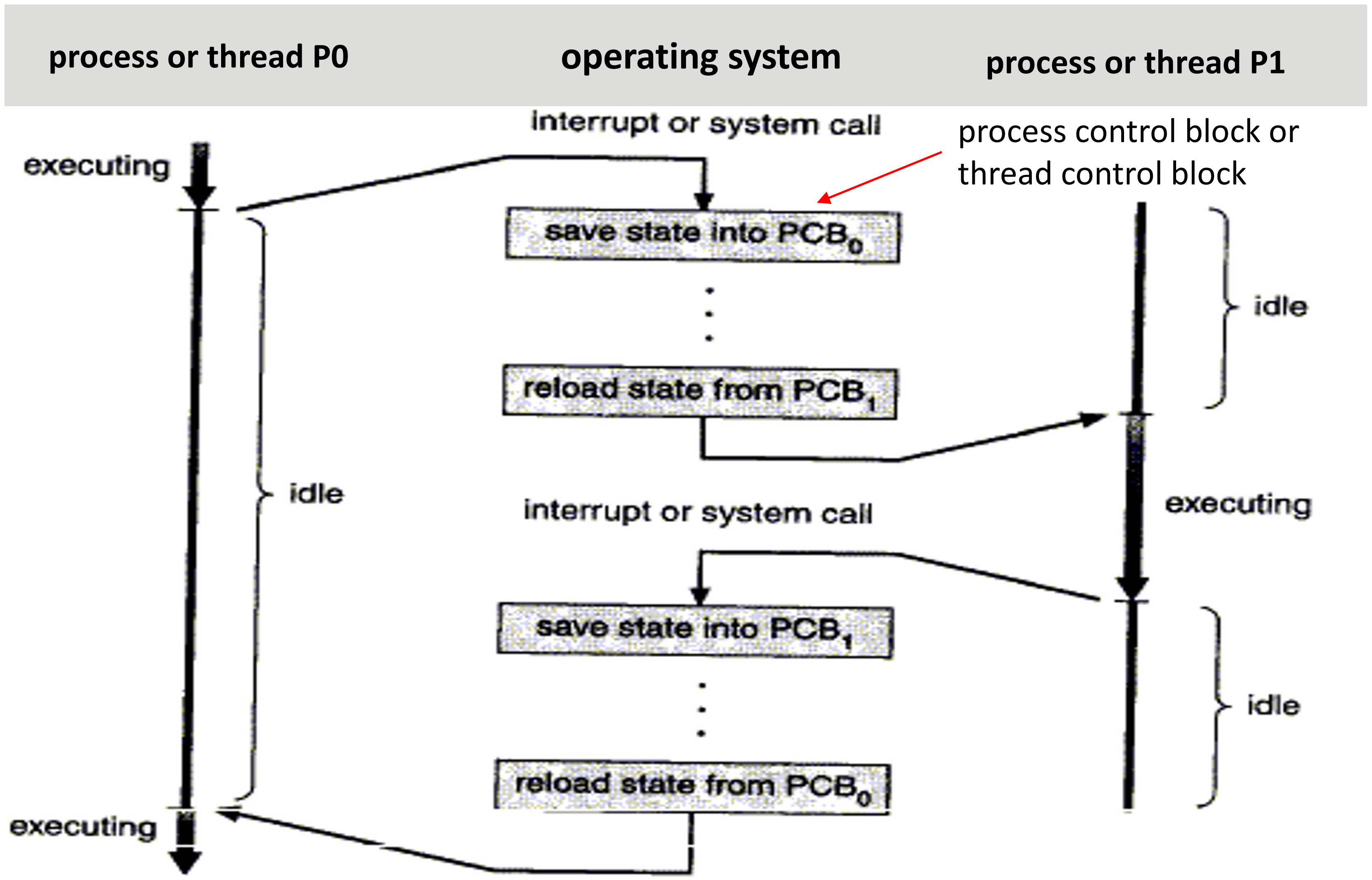
- *Thread advantages and characteristics*:
  - Faster to switch between threads than to switch between processes. Switching between user-level threads requires no major intervention by the OS.
  - Typically, an application will have a separate thread for each distinct activity.
  - The OS maintains for each thread a *Thread Control Block (TCB)* that stores all information needed to manage and schedule a thread. This includes the name of the thread, its priority and current state (e.g., program counter, scheduling info).

Thread Control Blocks (TCBs)

- The operating system manages TCBs using linked lists. Below is an example.



Context Switch: Processes or Threads



Embedded Operating Systems

Classes of Operating Systems

Class 1: Fast and Efficient Kernels

Fast and efficient kernels

For hard real-time systems, these kernels are questionable, because they are designed to be *fast* rather than *predictable* in every respect.

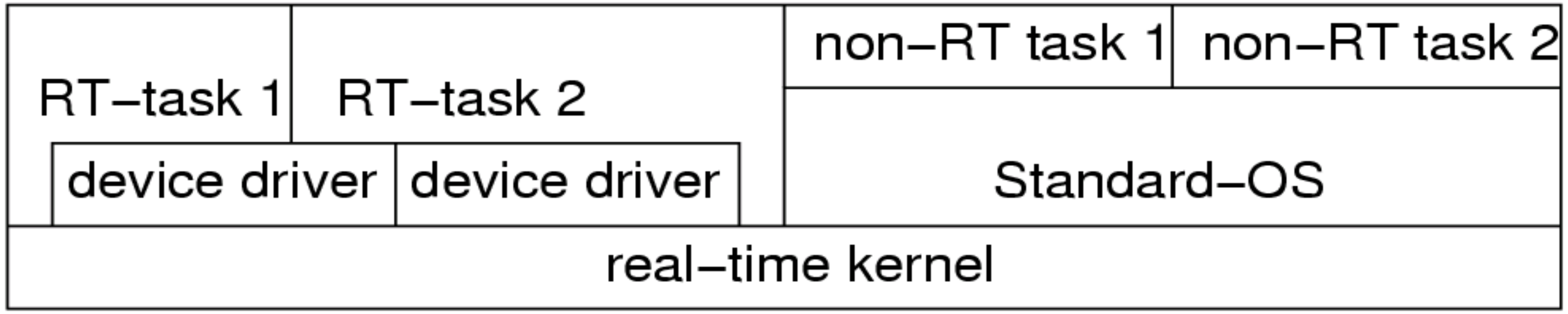
Examples include

FreeRTOS, QNX, eCOS, VxWORKS, LynxOS.

Class 2: Extensions to Standard OSs

Real-time extensions to standard OS:

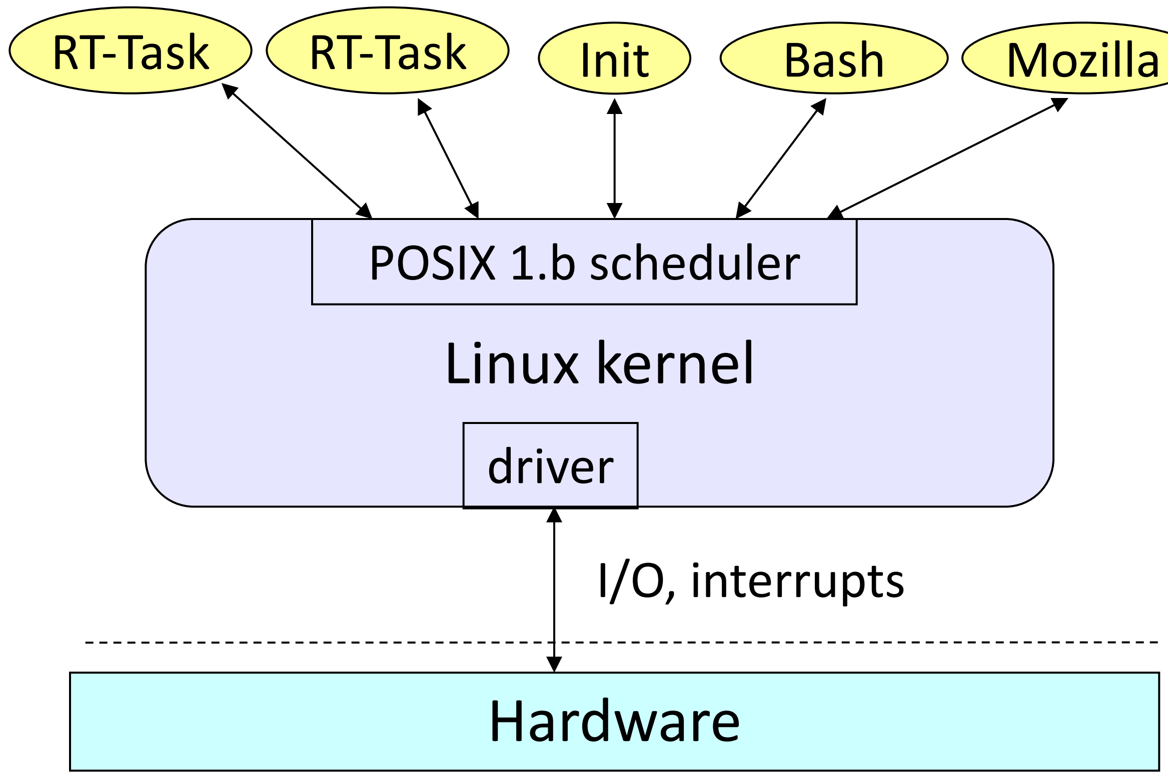
- Attempt to exploit existing and comfortable main stream operating systems.
- A real-time kernel runs all real-time (RT) tasks.
- The standard OS is executed as one task.



+ Crash of standard OS does not affect RT tasks
- RT tasks cannot use standard OS services
  (→ less comfortable than expected)

Example: Posix 1.b RT-extensions to Linux

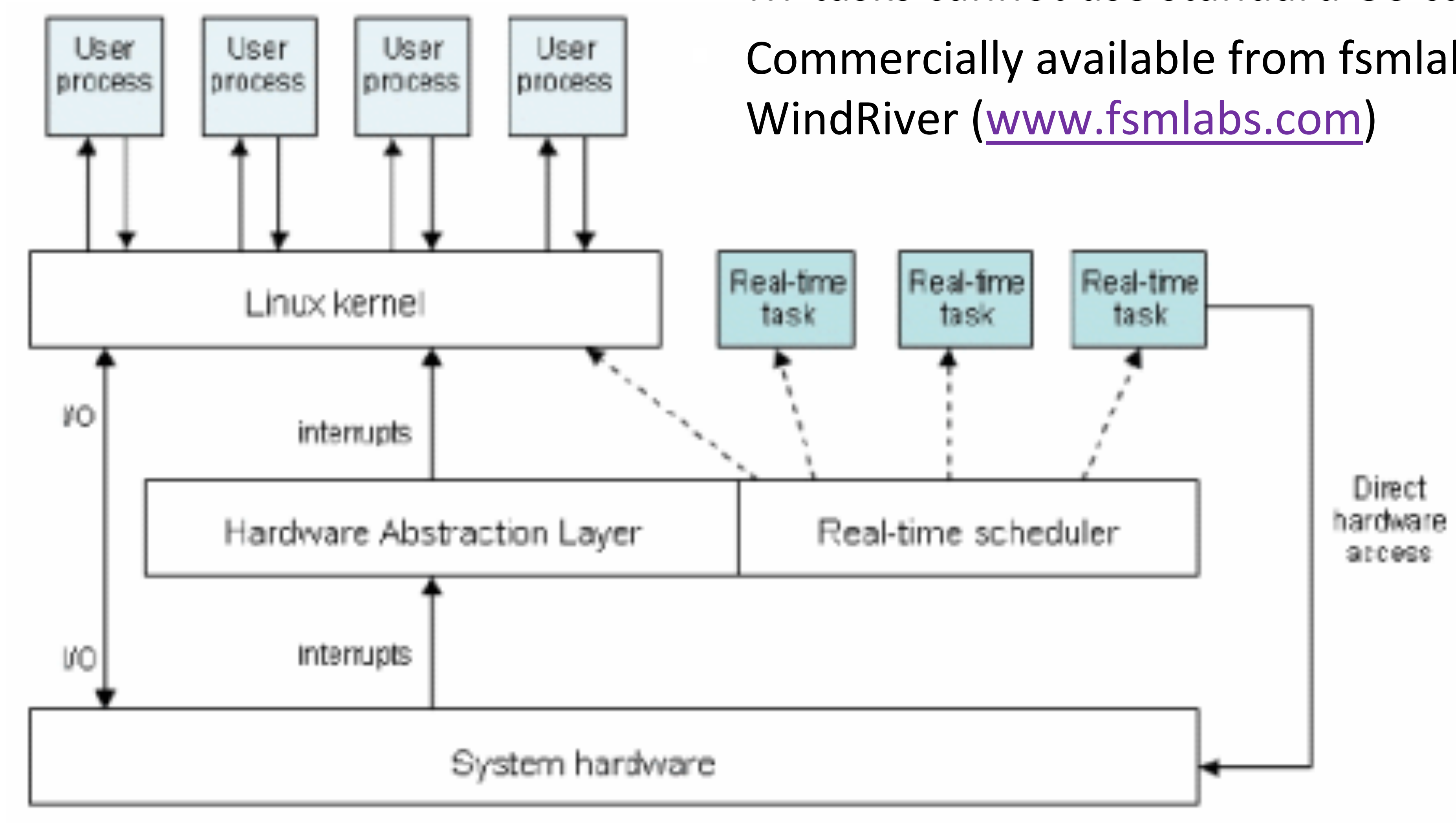
The standard scheduler of a general-purpose operating system can be replaced by a scheduler that exhibits *(soft) real-time properties*.



Special calls for real-time as well as standard operating system calls available.

Simplifies programming, but no guarantees for meeting deadlines are provided.

Example: RT Linux



RT tasks cannot use standard OS calls.

Commercially available from fsmlabs and WindRiver (www.fsmlabs.com)

Class 3: Research Systems

Research systems try to avoid limitations of existing real-time and embedded operating systems.

- Examples include L4, seL4, NICTA, ERIKA, SHARK

Typical research questions:

- low overhead memory protection
- temporal protection of computing resources
- RTOS for on-chip multiprocessors
- quality of service (QoS) control (besides real-time constraints)
- formally verified kernel properties

List of current real-time operating systems:
http://en.wikipedia.org/wiki/Comparison_of_real-time_operating_systems