Embedded Operating Systems

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:

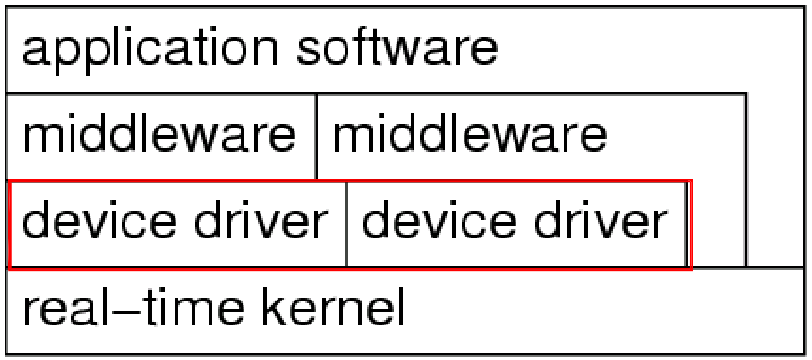
- *Remove unused functions*/libraries (for example by the linker).
- *Use conditional compilation* (using #if and #ifdef commands in C, for example).
- But deriving a consistent configuration is a potential problem of systems with a large number of derived operating systems. There is the danger of missing relevant components.

Embedded Operating System

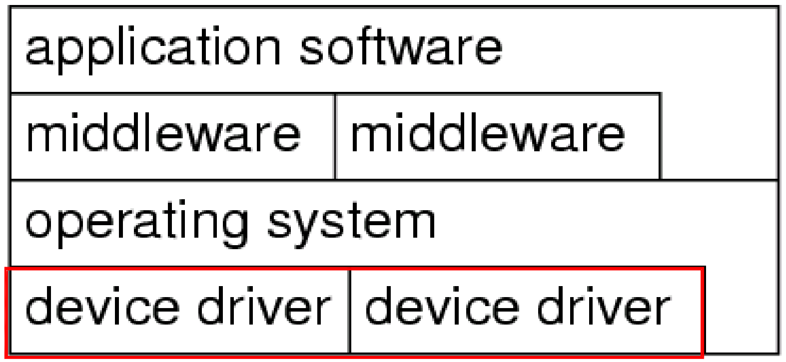
Device drivers are typically handled directly by tasks instead of drivers that are managed by the operating system:

- This architecture *improves timing predictability* as access to devices is also handled by the scheduler
- 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



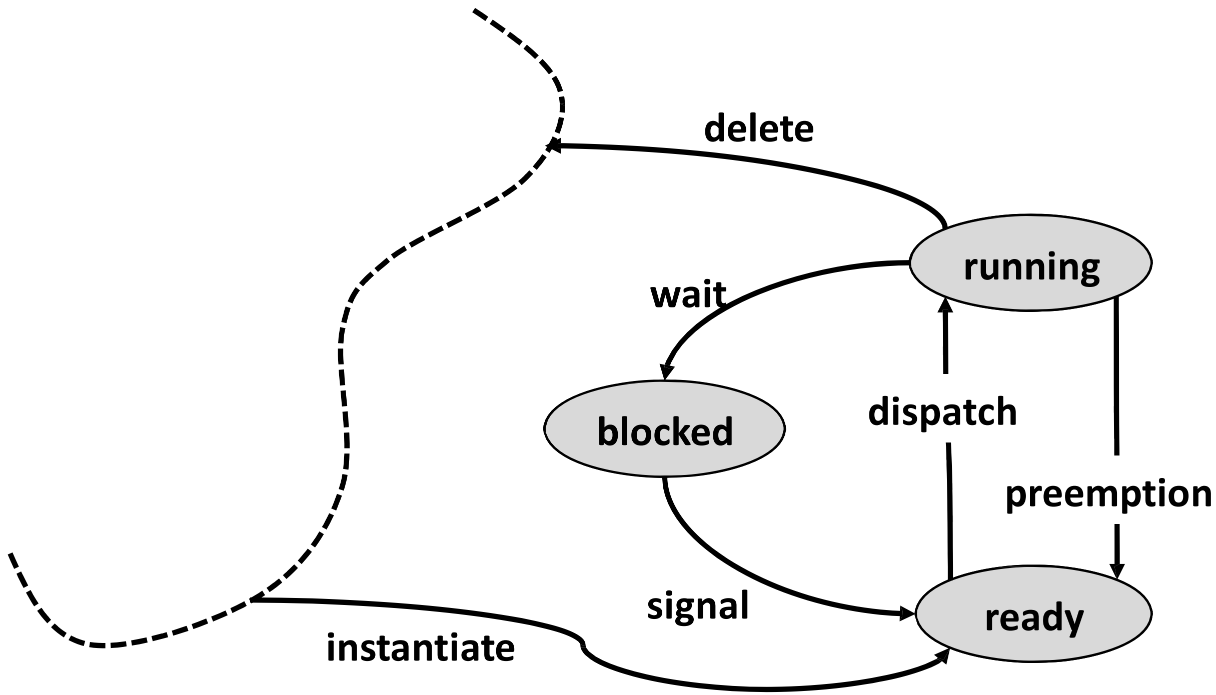
Main Functionality of RTOS-Kernels

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 is performed via semaphores and monitors.
  - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (for example 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 as 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 has the state “running”.

Blocked:

- A task enters the blocked state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore or timer. In this case, the task is inserted in a queue associated with this semaphore. The task at the head is resumed when the semaphore is unlocked by an event.