Real-Time Operating Systems With Example PICOS18

Sebastian Fischmeister

1

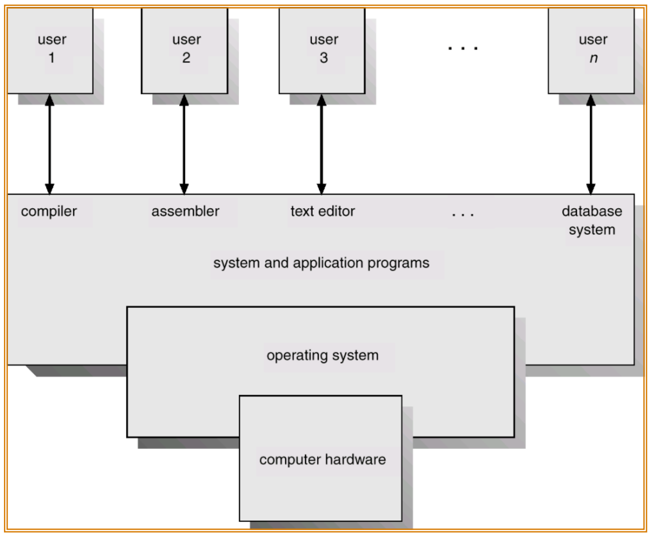
What is an Operating System?

- A program that acts as an intermediary between a user of a computer and the computer hardware
- Operating system goals:
  - o Execute user programs and make solving user problems easier.
  - o Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Components

1. Hardware – provides basic computing resources (CPU, memory, I/O devices)
2. Operating system – controls and coordinates the use of the hardware among the various application programs for the various users
3. Applications programs – define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs)
4. Users (people, machines, other computers)

Abstract View of System Components

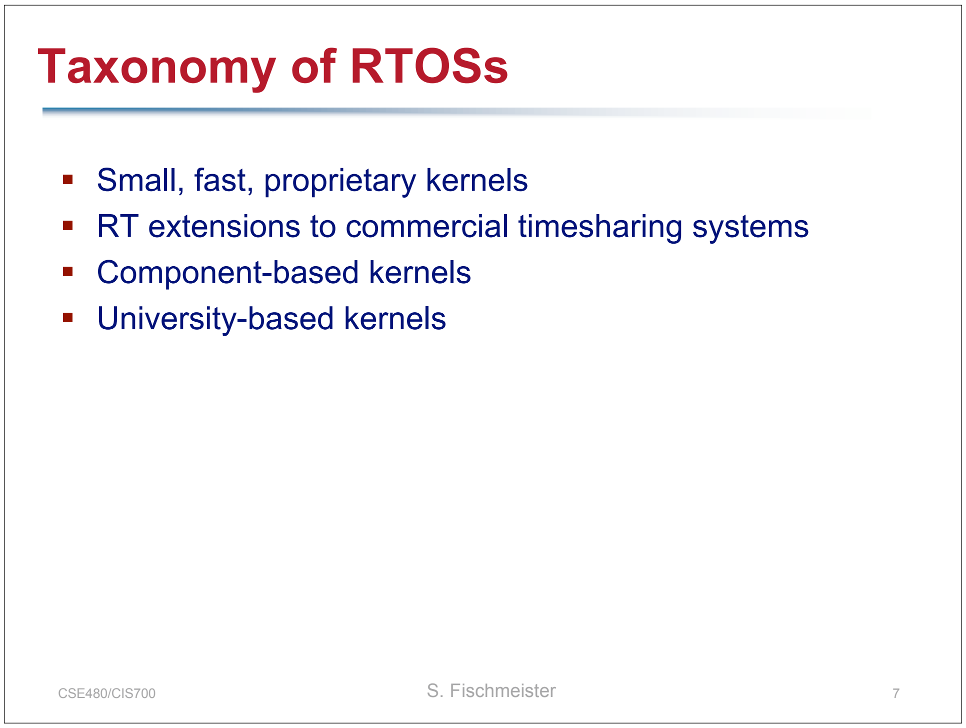


What is an RTOS?

- Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems
- Well-defined fixed-time constraints

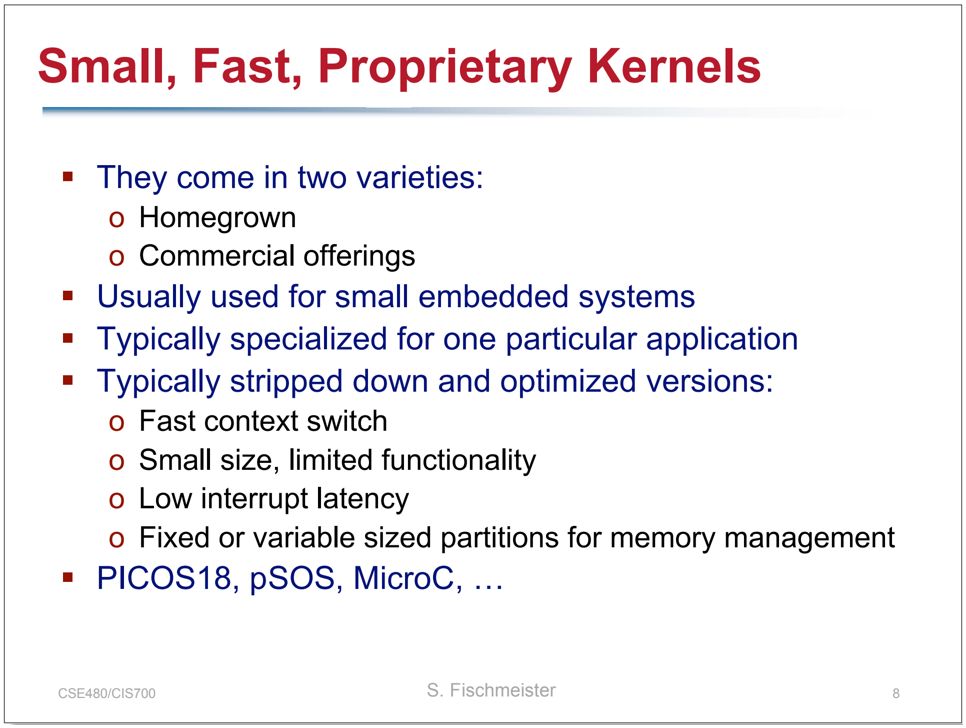
More Precisely?

- The system allows access to sensitive resources with defined response times.
  - Maximum response times are good for hard real-time
  - Average response times are ok for soft real-time
- Any system that provides the above can be classified as a real-time system
  - 10us for a context switch, ok?
  - 10s for a context switch, ok?



Taxonomy of RTOSs

- Small, fast, proprietary kernels
- RT extensions to commercial timesharing systems
- Component-based kernels
- University-based kernels



Small, Fast, Proprietary Kernels

- They come in two varieties:
  - Homegrown
  - Commercial offerings
- Usually used for small embedded systems
- Typically specialized for one particular application
- Typically stripped down and optimized versions:
  - Fast context switch
  - Small size, limited functionality
  - Low interrupt latency
  - Fixed or variable sized partitions for memory management
- PICOS18, pSOS, MicroC, …

RT Extensions

- A common approach is to extend Unix
  - Linux: RT-Linux, RTLinuxPro, RTAI,
  - Posix: RT-Posix
  - MACH: RT-MACH
- Also done for Windows based on virtualization.
- Generally slower and less predictable.
- Richer environment, more functionality.
- These systems use familiar interfaces, even standards.
- Problems when converting an OS to an RTOS:
  - Interface problems (nice and setpriority in Linux)
  - Timers too coarse
  - Memory management has no bounded execution time
  - Intolerable overhead, excessive latency

How to do an RT Extension?

- Compliant kernels
  - Takes an existing RTOS and make it execute other UNIX binaries (see LynxOS).
  - Interfaces need to be reprogrammed.
  - Behavior needs to be correctly reimplemented.

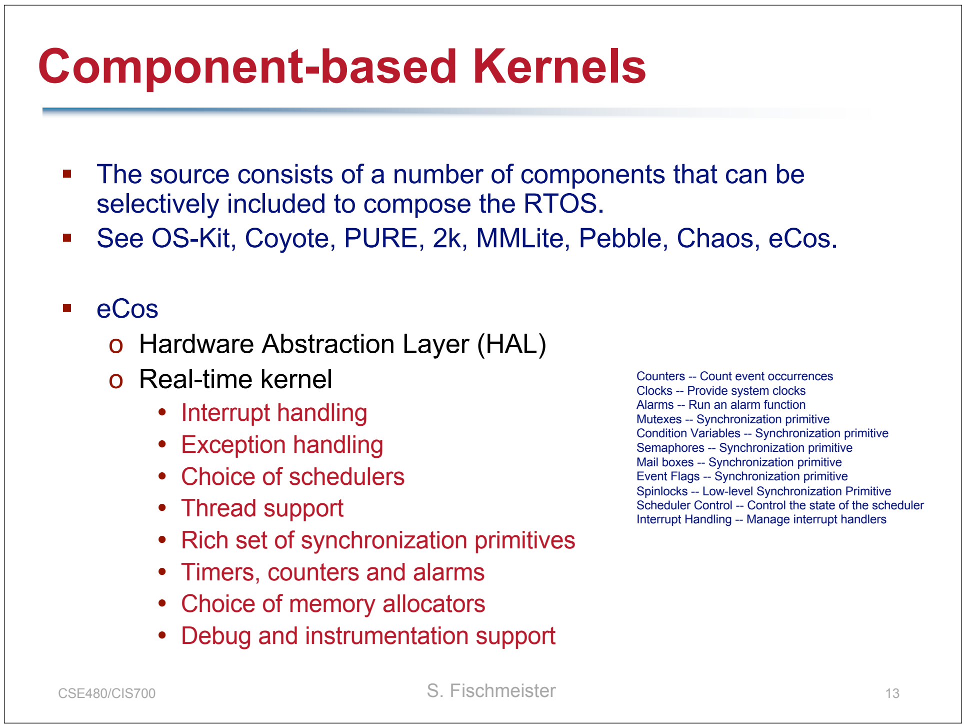
How to do an RT Extension?

- Dual kernels
  - o Puts an RTOS kernel between the hardware and the OS.
  - o Hard tasks run in the RTOS kernel, the OS runs when CPU is available.
  - o Native applications can run without any changes.
  - o Hard tasks get real-time properties.
  - o See RTLinuxPro

- Problems:
    - A single failing hard task can kill the whole system.
    - The RTOS kernel requires its own IO drivers.

How to do an RT Extension?

- Core kernel modifications
  - o Takes the non-RT operating systems and modifies it to become an RTOS.
- Problem: (need to do all this)
  - o Implement high-resolution timers
  - o Make the kernel preemptive
  - o Implement priority inheritance
  - o Replace FIFOs with priority queues
  - o Find and change long kernel execution paths

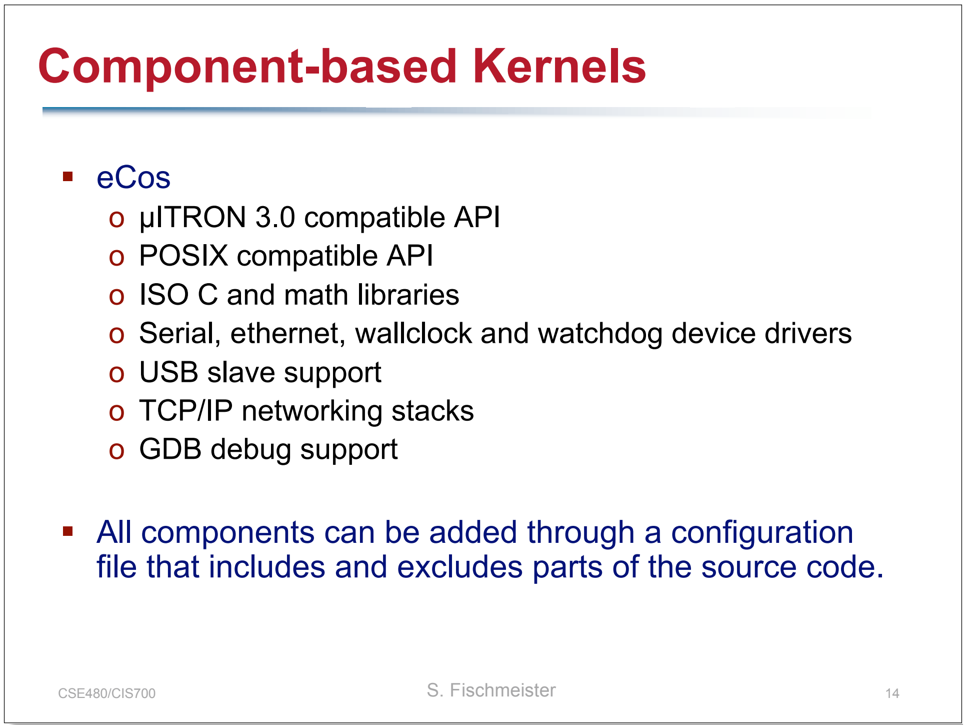


Component-based Kernels

- The source consists of a number of components that can be selectively included to compose the RTOS.
- See OS-Kit, Coyote, PURE, 2k, MMLite, Pebble, Chaos, eCos.

- eCos
  - Hardware Abstraction Layer (HAL)
  - Real-time kernel
    - Interrupt handling
    - Exception handling
    - Choice of schedulers
    - Thread support
    - Rich set of synchronization primitives
    - Timers, counters and alarms
    - Choice of memory allocators
    - Debug and instrumentation support

Counters -- Count event occurrences
Clocks -- Provide system clocks
Alarms -- Run an alarm function
Mutexes -- Synchronization primitive
Condition Variables -- Synchronization primitive
Semaphores -- Synchronization primitive
Mail boxes -- Synchronization primitive
Event Flags -- Synchronization primitive
Spinlocks -- Low-level Synchronization Primitive
Scheduler Control -- Control the state of the scheduler
Interrupt Handling -- Manage interrupt handlers



Component-based Kernels

- eCos
  - µITRON 3.0 compatible API
  - POSIX compatible API
  - ISO C and math libraries
  - Serial, ethernet, wallclock and watchdog device drivers
  - USB slave support
  - TCP/IP networking stacks
  - GDB debug support

- All components can be added through a configuration file that includes and excludes parts of the source code.

Research Kernels

- Many researchers built a new kernel for one of these reasons:
  - Challenge basic assumptions made in timesharing OS
  - Developing real-time process models
  - Developing real-time synchronization primitives
  - Developing solutions facilitating timing analysis
  - Strong emphasis on predictability
  - Strong emphasis on fault tolerance
  - Investigate the object-oriented approach
  - Real-time multiprocessor support
  - Investigating QoS

What Typically Differs

Requirements

- RTOS must be predictable
  - We have to validate the system
  - We have to validate the OS calls/services
- We must know upper bounds to
  - Execution time of system calls
  - Memory usage
- We must have static bounds on
  - Memory layout
  - Size of data structures (e.g. queues)
- Fine grain interrupt control

RTOS Predictability

- All components of the RTOS must be predictable
  - System calls, device drivers, kernel internal management
- Memory access
  - Page faults, lookup time, caches
- Disk access
  - Bound for head movement while reading/writing data
- Net access
  - Bound for time for transmission, switching
  - Dropped packets??
- Scheduling must be deterministic

Admission Control

- Admission control is a function that decides if new work entering the system should be admitted or not.
- To perform this it requires:
  - A model of the state of system resources
  - Knowledge about incoming requests
  - An algorithm to make the admission decision
  - Policies for actions to take upon admission and rejection
- Statically scheduled systems require no admission control.

Admission Control

- The admission algorithm requires preanalyzed tasks
  - Shared data
  - Execution time
  - Precedence information
  - Importance level
  - Deadlines
- Positive decision assigns time slices to the task
- Negative decision has options:
  - Run a simpler version of the task
  - Run on a different machine
  - Reject the task
- Admission algorithms can be complex as they have to consider multiple resources (e.g., networked video streaming).

Resource Reservation

- Resource reservation is the act of actually assigning resources to a task.
  - Initially no resource reservation, only allocation as the task runs.
  - Valuable for hard real-time systems.
  - Introduces an overhead as resources might be unused
    - => introduction of resource reclaiming strategies
- Closely linked to resource kernels that offer interfaces for resource reservation, donation, and reflection.

Task Declaration

- RTOSs tailored to microprocessors often require a static declaration of tasks.
- Advantages are:
  - Simple check that the system has sufficient resources.
  - No admission control necessary.
  - No overhead introduced by the admission test.
  - No thread spawning problems
- => but quite static

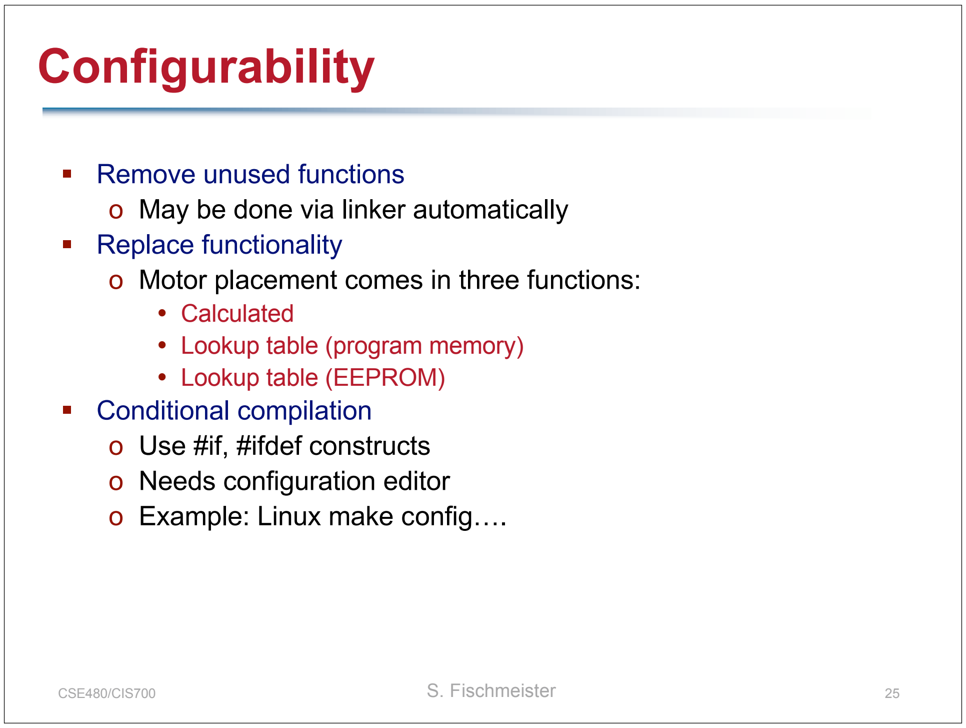
Boot from ROM

- The RTOS typically boots from the ROM when used on microprocessors.
- Requires the application program to actually start up the RTOS:

```
void main (void) {
   /* Perform Initializations */
  ...
  OSInit();
  ...
   /* Create at least one task by calling
    OSTaskCreate() */
  OSStart();
}
```

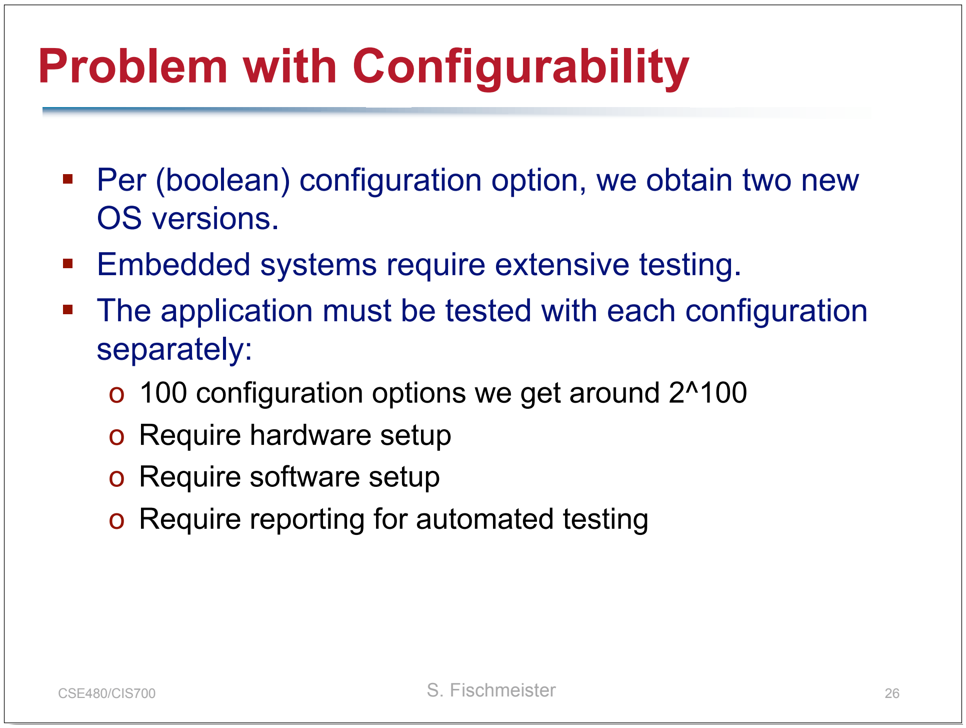
Configurability

- As mentioned with component-based RTOS, the system must be configurable.
- Include only components needed for the present system
- Components must be removable
  - Inter-module dependencies limit configurability
- Configuration tailors OS to system
  - Different configuration possibilities
- Example RoboVM (PICDEM and modular robot).



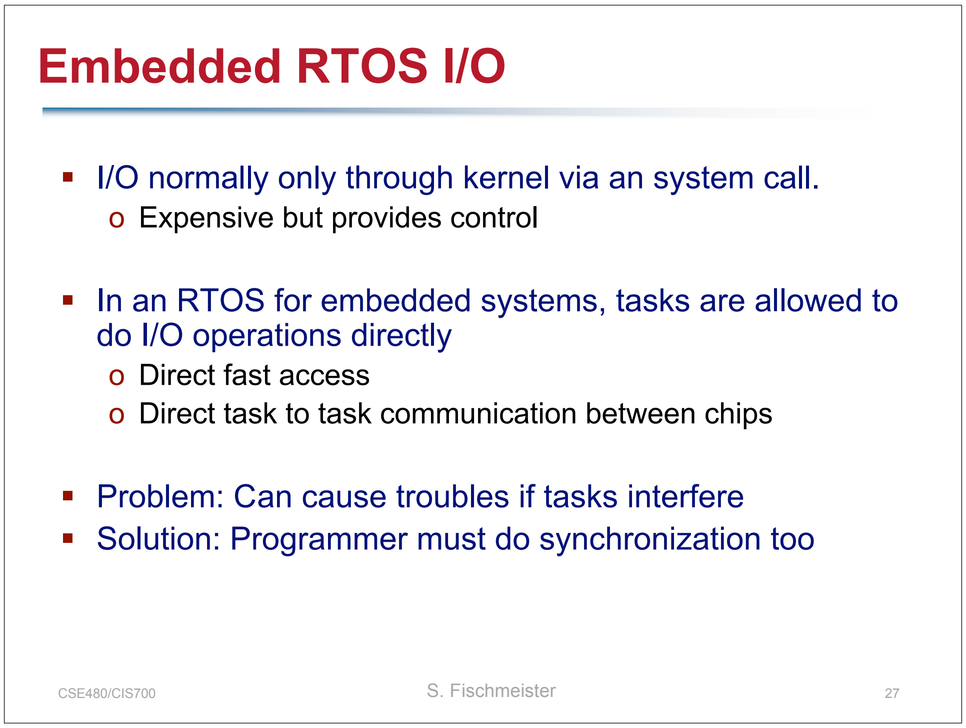
Configurability

- Remove unused functions
  - May be done via linker automatically
- Replace functionality
  - Motor placement comes in three functions:
    - Calculated
    - Lookup table (program memory)
    - Lookup table (EEPROM)
- Conditional compilation
  - Use #if, #ifdef constructs
  - Needs configuration editor
  - Example: Linux make config….



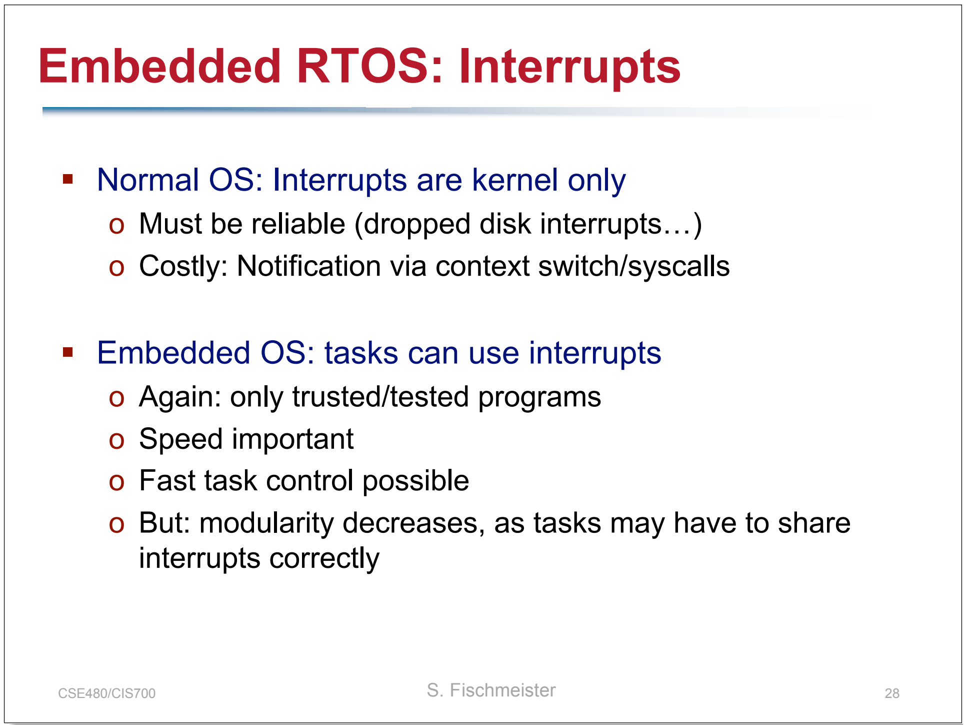
Problem with Configurability

- Per (boolean) configuration option, we obtain two new OS versions.
- Embedded systems require extensive testing.
- The application must be tested with each configuration separately:
  - 100 configuration options we get around 2^100
  - Require hardware setup
  - Require software setup
  - Require reporting for automated testing



Embedded RTOS I/O

- I/O normally only through kernel via an system call.
  - Expensive but provides control
- In an RTOS for embedded systems, tasks are allowed to do I/O operations directly
  - Direct fast access
  - Direct task to task communication between chips
- Problem: Can cause troubles if tasks interfere
- Solution: Programmer must do synchronization too



Embedded RTOS: Interrupts

- Normal OS: Interrupts are kernel only
  - Must be reliable (dropped disk interrupts…)
  - Costly: Notification via context switch/syscalls
- Embedded OS: tasks can use interrupts
  - Again: only trusted/tested programs
  - Speed important
  - Fast task control possible
  - But: modularity decreases, as tasks may have to share interrupts correctly

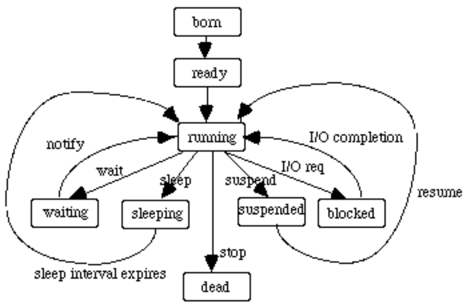
PICOS18

Terminology

- Critical section, or critical region, is code that needs to be treated indivisibly.
  - No interrupts
  - No context switch
- Resource is an entity used by a task.
  - Printer, keyboard, CAN bus, serial port
- Shared resource is a resource that can be used by more than one task.
  - => mutual exclusion
- Multitasking is the process of scheduling and switching the CPU between several tasks.

Task

- Task, also called thread, is a user application.
  - Shares the CPU and resources with other tasks
  - Follows a defined life cycle



Context Switches

- A context switch occurs whenever the multitasking kernel decides to run a different task.
  - Save the current task’s context in the storage area.
  - Restores the new task’s context from the storage area.
  - Resumes the new task

- Context switching adds overhead.
- The more registers a processor has, the higher the overhead => irrelevant for RTOS as long as its known.

Kernels

- The kernel is responsible for managing the tasks.
- Most fundamental service is the context switch.

- Non-preemtive kernels, also cooperative multitasking
  - The task needs to explicitly give up control of the CPU.
  - Allows low interrupt latency, because they may be never disabled.
  - Allows non-reentrant functions at the task level.
  - Response time is determined by the longest task.
  - No overhead for protecting shared data.
  - Responsiveness may be low, because of low priority task requiring a lot of time until it releases the CPU.

Kernels

- Preemptive kernel
  - Responsiveness is good, because tasks get preempted.
  - A higher-priority task can preempt a lower priority task that still requires more time to compute.
  - Response time becomes deterministic, because at the next tick, the OS switches to the other new task.
  - Non-reentrant functions require careful programming.
  - Periodic execution of the 'tick' adds to the overhead.

Introduction

- PICOS18 is a preemptive RTOS for the PIC18 series.
- Bases on OSEK/VDX, an open industry standard.
- Developed by Pragmatec.
- GPL

- www.picos18.com
- www.picos18.com/forum

Services

- PICOS18 provides
  - Core services: initialization, scheduling
  - Alarm and counter manager
  - Hook routines
  - Task manager
  - Event manager
  - Interrupt manager

PICOS18 Interrupt Routine

- Part of the user application.
- One for the high priority interrupts and one for low priority interrupts.
- Most important part: *AddOneTick()*
- *Let’s have a look.*

PICOS18 Context Switch

- The active task gets suspended and its context gets pushed onto its stack.
- The preempted task gets resumed and its context gets restored.
- *Let’s have look at the save_task_ctx routine.*

Static Declarations

- PICOS18 requires you to statically declare
  - Alarms
  - Resources
  - Tasks
- *Let’s have a look.*

Task API

- StatusType ActivateTask (TaskType TaskID)
  - Change the state of a task from SUSPENDED to READY.
- StatusType TerminateTask (void)
  - Changes the state of a task from READY to SUSPENDED.
- StatusType ChainTask (TaskType TaskID)
  - Terminates the current task, activates a follow up task.
- StatusType Schedule(void)
  - Invoke the scheduler to find a new active task.
  - Not necessary, because PICOS18 is a preemptive OS.
- StatusType GetTaskID (TaskRefType TaskID)
- StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)

Tasks Implementation

- At most 16 events.
- The task state is encoded in the following variables:
  - tsk_X_state_ID
    - Bits 0-3: task identifier
    - Bit 4: unused
    - Bit 5-7: task state
  - tsk_X_active_prio
    - Bits 0-3: task priority
    - Bit 5-7: activation counter

  - Let's look at some of the functions in pro_man.c

Event Management

- StatusType SetEvent (TaskType TaskID, EventMaskType Mask)
  - Posts an event to another task. Causes a scheduling operation.
- StatusType ClearEvent (EventMaskType Mask)
  - Clears the event, otherwise an infinite loop.
- StatusType GetEvent (TaskType TaskID, EventMaskRefType Event)
  - Receives the event value for a specific task.
- StatusType WaitEvent (EventMaskType Mask)
  - Blocks the current task until the event occurs.

Event Implementation

- At most 16 events.
- The event status is encoded in these two variables:
  - EventMaskType event_X
    - For each task 16 possible events.
  - EventMaskType wait_X
    - Each task can listen for 16 possible events.
- *Let's have a look at the code.*

Alarm Management

- StatusType GetAlarm (AlarmType AlarmID, TickRefType Tick)
  - Returns the number of ticks until the alarm goes off.
- StatusType SetRelAlarm (AlarmType AlarmID, TickType increment, TickType cycle)
  - Registers an alarm relative to the current kernel counter.
- StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)
  - Registers an alarm as absolute kernel counter tick value.
- StatusType CancelAlarm (AlarmType AlarmID)
  - Deactivate an alarm.

Alarm Implementation

- Each tick the alarm counters get incremented by one.
- If the alarm value equals the counter value, then the alarm will cause an event.
- *Let's look at the code.*

Sample Application

- *Let's look at the sample application that comes with PICOS18.*