



Industrial Process Controllers and Simulators

Topic 5

Real-Time software environment

Real-Time Software Environment

1. The operational environment
2. The kernel
3. Tasks
4. ISR-task





Complex Real-Time Systems

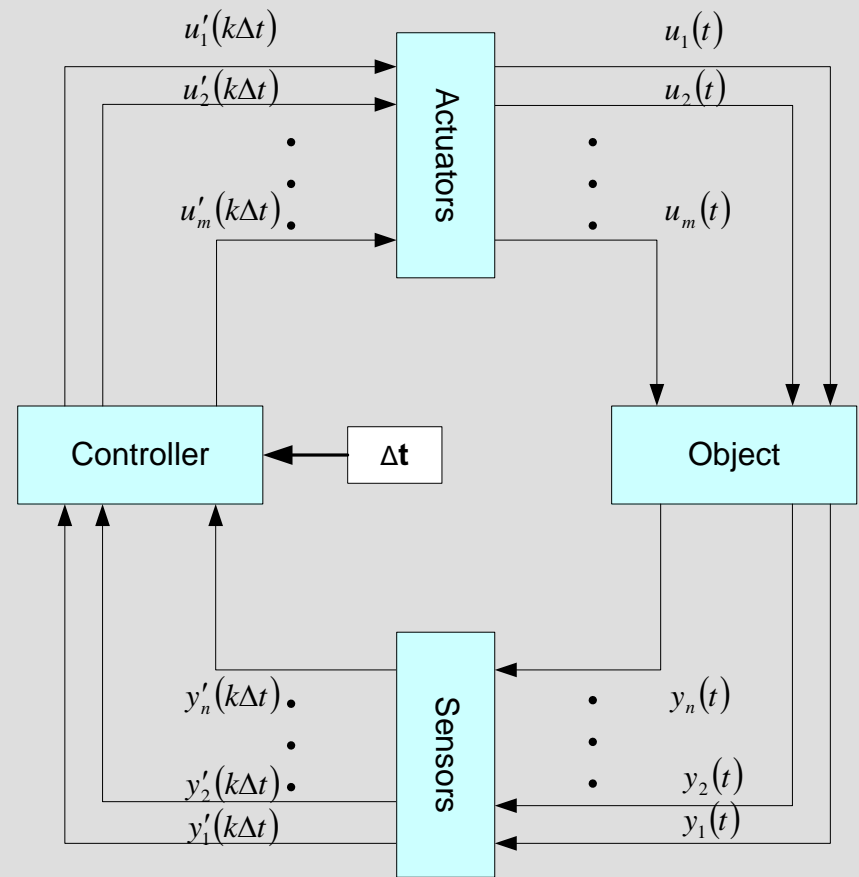
1. **Basic structure**

2. **Real-Time Operating Systems**

Complex Real-Time Systems

Basic structure:

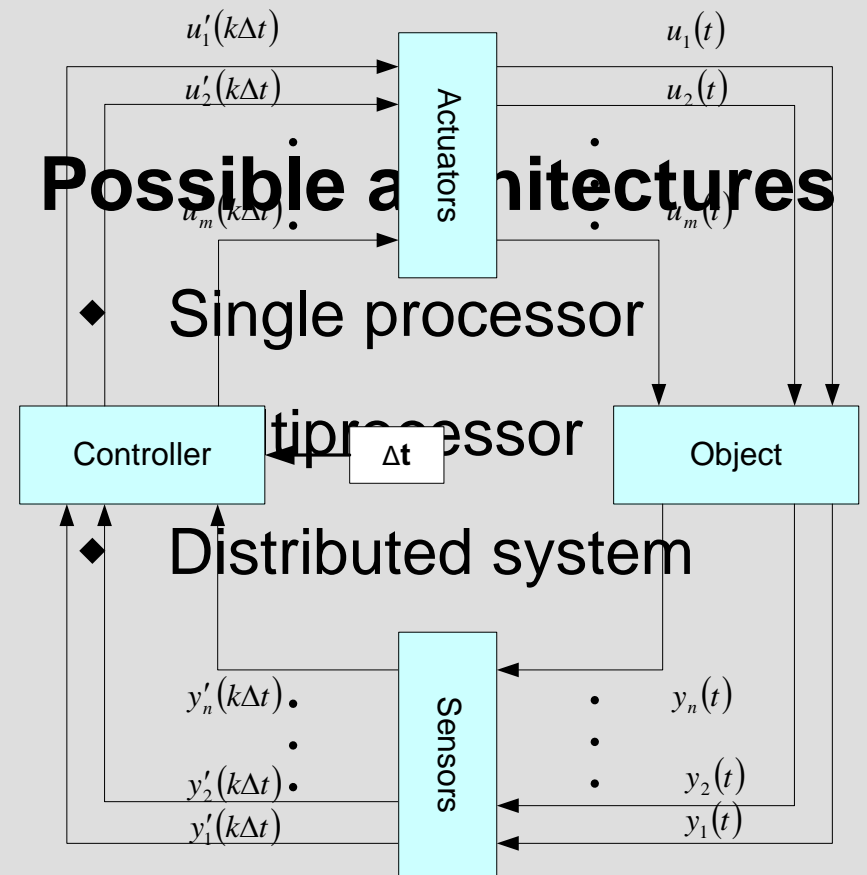
1. A controlling system
2. A controlled system
3. The environment



Complex Real-Time Systems

What is the problem?

- ◆ Have some work to do
 - know subtasks
- ◆ Have limited resources
- ◆ Have some constraints to meet
- ◆ Want to optimize quality



Complex Real-Time Systems

Why do we need operating system?

- In many cases we don't need it
 - ◆ one or two control loops / automata
 - ◆ no user interface
 - ◆ no or very simple communication
- In complex cases we need it
 - ◆ many computational tracks
 - ◆ complex user interface
 - ◆ real-time and / or general purpose communication
 - ◆ needs to isolate computation processes

Complex Real-Time Systems

What is an Operating System ?

Provides environment for executing programs:

- The main question is :
- ◆ Process abstraction for multitasking/concurrency
 - ◆ Scheduling
 - ◆ Hardware abstraction layer (device drivers)
 - ◆ File systems
 - ◆ Communication

Our focus is concurrency and real-time issues

Real-Time Operating Systems

Definition:

“A Real-Time system is the one in which the correctness of the output depends not only on the logical results, but also on the time at which results are produced.”

Real-Time Operating Systems

Main goal of an RTOS scheduler

meeting deadlines

Correct Output = Correct result + Correct Time

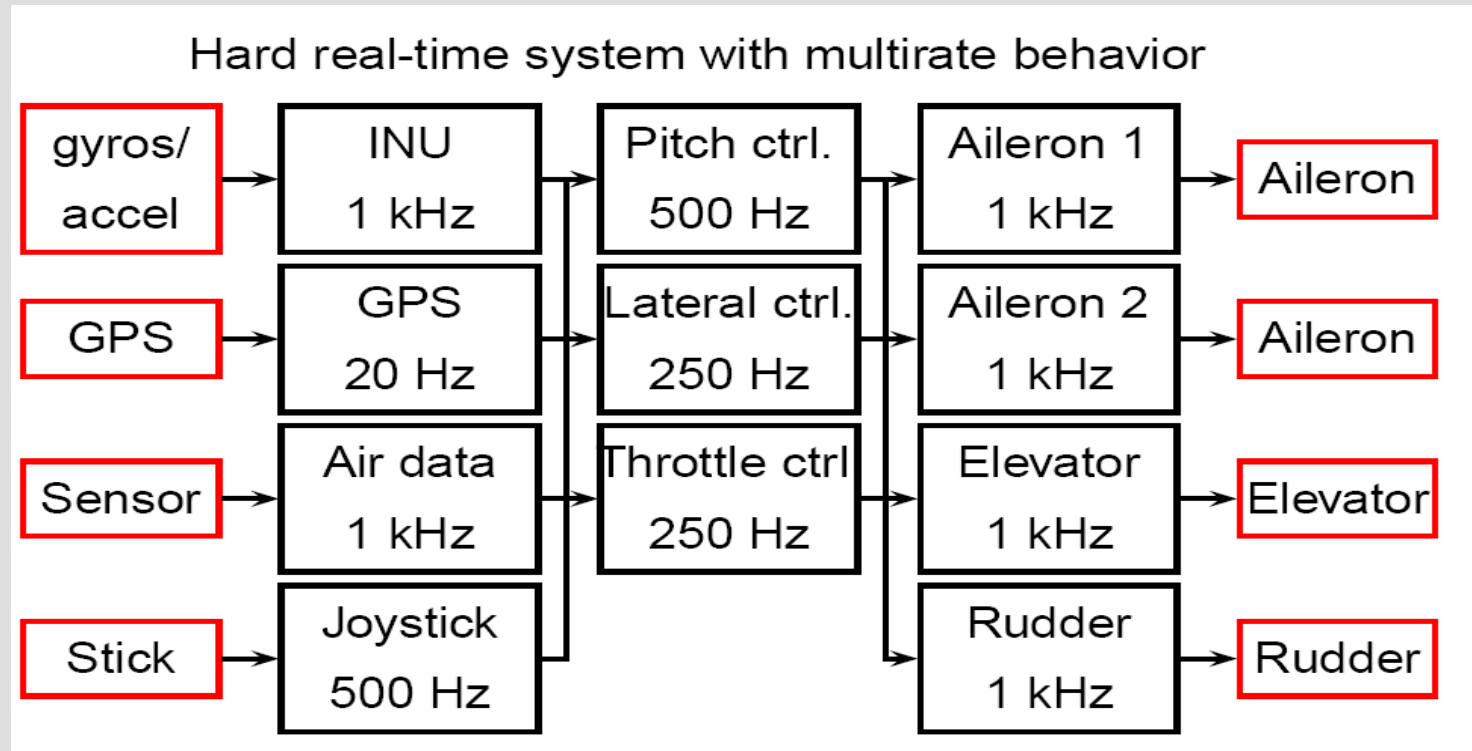
Real-Time Operating Systems

For engineers :

- ◆ A system where correctness depends not only on the correctness of the logical result of the computation, but also on the result delivery time.
- ◆ A system that responds in a timely, predictable way to unpredictable external stimuli arrivals.
- ◆ *RT OS* is *NOT* a transactional system.
- ◆ *RT OS* is one that has a bounded (predictable) behavior under all system load scenarios
 - It is just a *building block* – it does not guarantee system correctness

Real-Time Operating Systems

Example:



Real-Time Operating Systems

Types of *RT OS*:

- ◆ Hard Real-Time
- ◆ Firm Real-Time
- ◆ Soft Real-Time
- ◆ Non Real-Time

Missing a deadline has catastrophic results for the system

Missing a deadline an unacceptable quality reduction

- Reduction in system quality is acceptable

No deadlines have to be met
can be recovered from

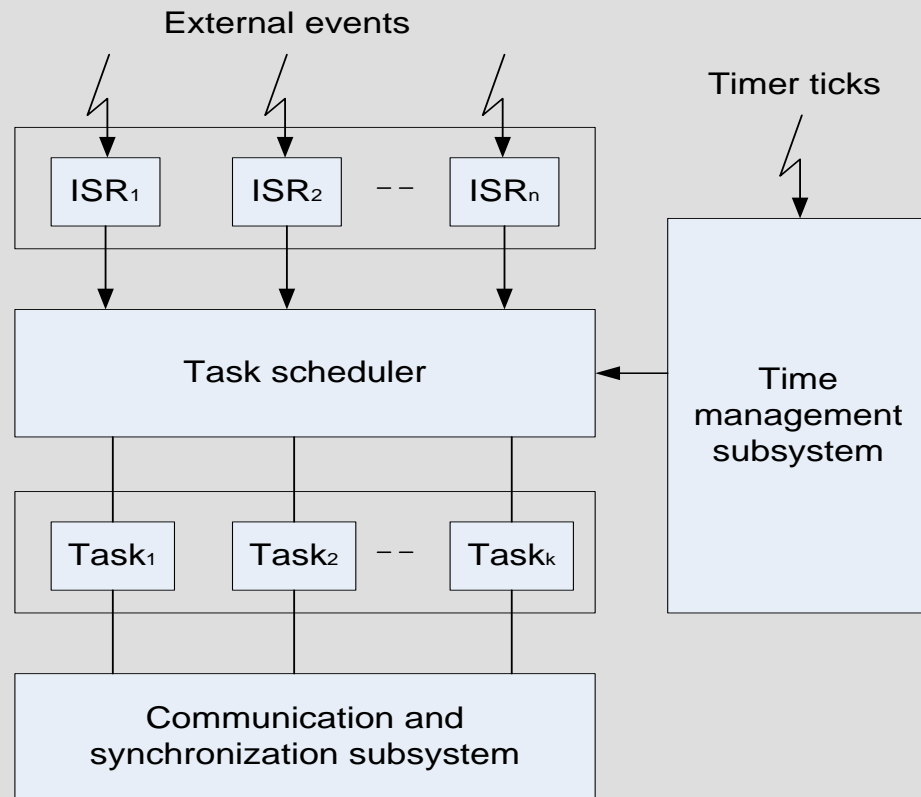
Real-Time Operating Systems

Real-Time Kernels:

- ◆ Monolithic kernel
 - ◆ Layered kernel
 - ◆ Distributed kernel
- a one piece of code
 - small size
 - separated to subsystems – object design is often presentation of distributed system as virtual unicomputer programs
 - more complicated but stable
 - bigger reaction time

Real-Time Operating Systems

Structure



Real-Time Operating Systems

Main tasks

- ◆ Process Management
- ◆ Interprocess Communication
- ◆ Memory Management
- ◆ Input/Output Management and Interrupt Handling

Real-Time Main Tasks

Input / Output Management

- ◆ Hardware adaptation level for devices' control
- ◆ Handles requests for read and write data for process and communication peripherals

Real-Time Operating System

Sporadic Real-Time Tasks

Aperiodic tasks with known minimum inter-arrival time.

This makes sporadic tasks *pseudoperiodic*.

Real-Time Operating System

Task constraints

- ◆ Deadline constraint
- ◆ Resource constraints
- ◆ Precedence constraints
- ◆ Fault-tolerant requirements

$T_1 \rightarrow T_2$:

Task T_2 can start executing only after T_1 finishes its

- To achieve higher reliability for task execution
- Redundancy in execution

Complex Real-Time Systems

Task comparison

	Hard Real-Time	Soft Real-Time
Response Time	Hard-required	Soft-required
Peak-load performance	Predictable	Degradable
Control of pace	Environment	Computer
Safety	Critical	Non-critical
Redundancy	Active	Checkpoint-recovery
Data integrity	Short-term	Long-term
Error-detection	Autonomous	Externally assisted

Kernel management functions

Main types:

- ◆ Kernel task management functions
- ◆ Kernel time-management functions
- ◆ External event processing functions

Kernel task management functions

Possible dispatched elements:

◆ Processes

◆ Threads

◆ Fibers

The thread is lightweight process.

The fiber is very *lightweight unit scheduled by application.*

The fiber has:

- *Code* -> only.

The fiber is not applicable in most RT OS.

Benefits of usage → pre-design and simulation of RT threads.

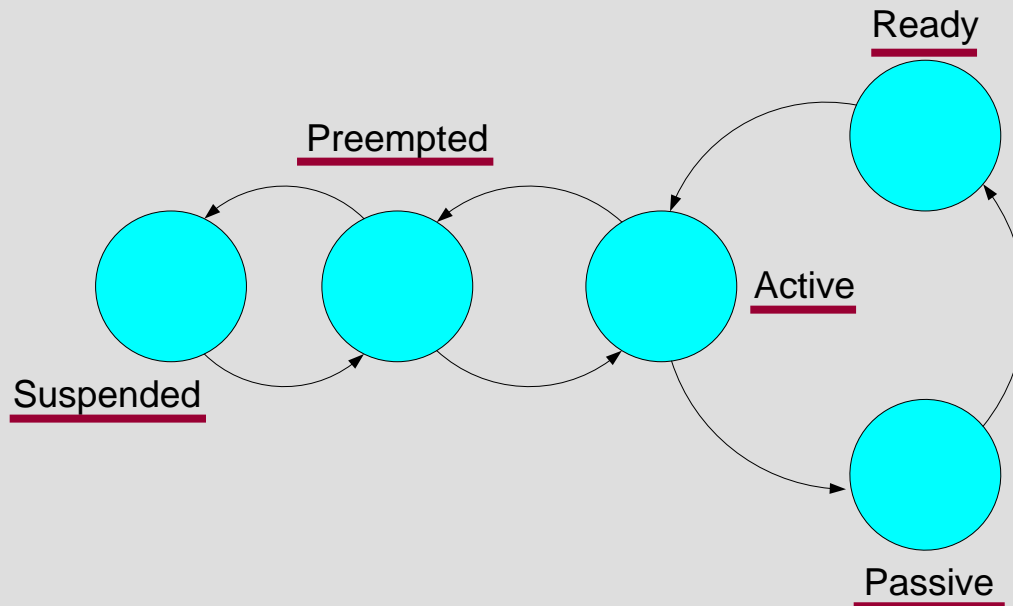
Kernel task management functions

Main purpose

scheduling CPU time

Kernel task management functions

Task status graph: tasks

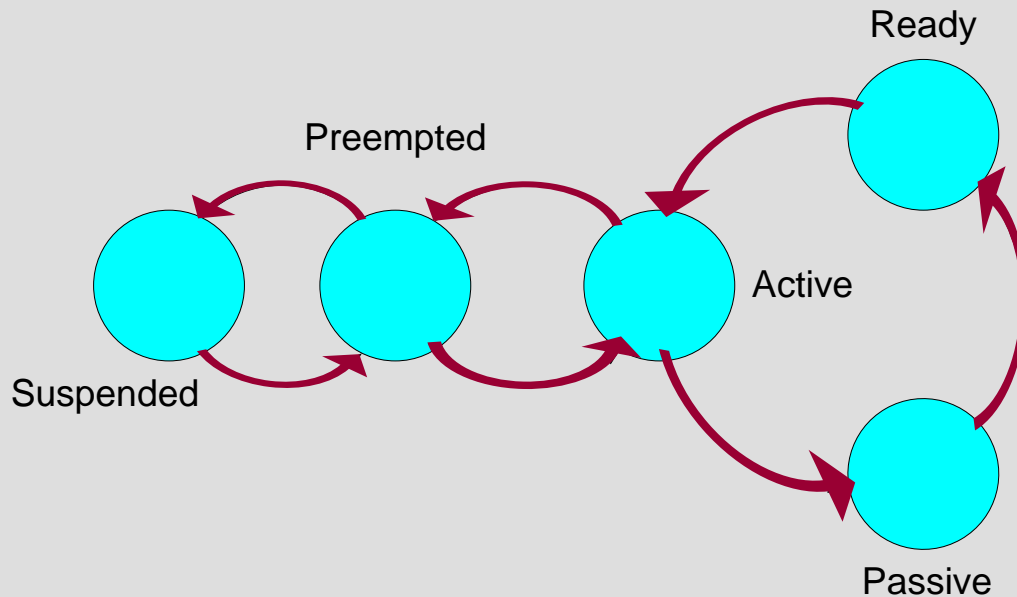


Ready Task:

the task is waiting for the processor to be available for execution for some reason

Kernel task management functions

Task status graph: change reasons



Reasons for a transition:

system (cpu) idle, no simple to finish, or 'schedule', 'quit', etc.)

Kernel task management functions

Task management disciplines

- ◆ **FIFO**
- ◆ **Round-robin**
- ◆ **Priority scheduling**

Task management disciplines

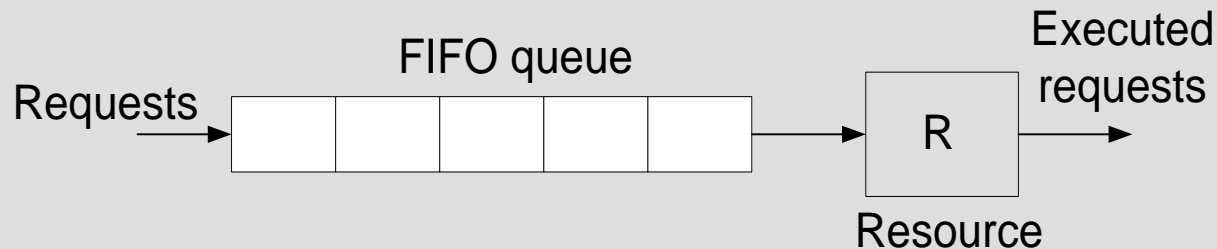
FIFO

The earliest known and the simplest discipline.

- ◆ Most applicable for batch processes
- ◆ Not applicable for real-time systems

The ready queue is a single FIFO queue where the next process to be run is the one at the front of the queue.

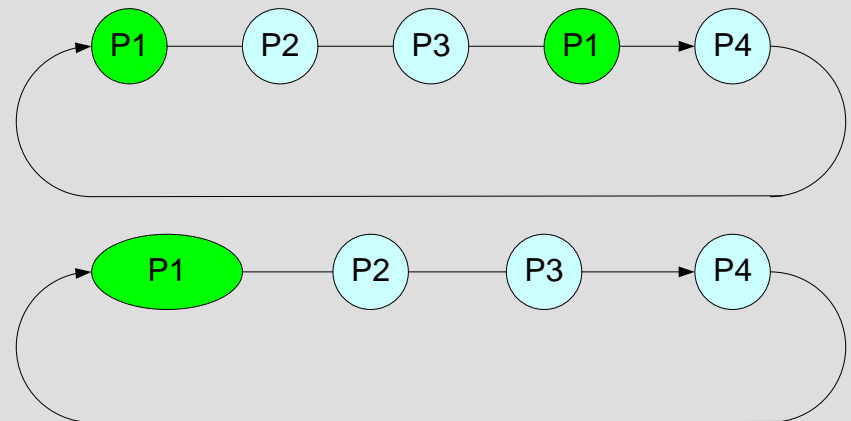
Processes are added to the back of the ready queue.



Task management disciplines

Round-robin - Weighted

Weighted round-robin is oriented to assign CPU to different tasks in non-equal manner. Thus primitive type of prioritized scheduling is realized.



Task management disciplines

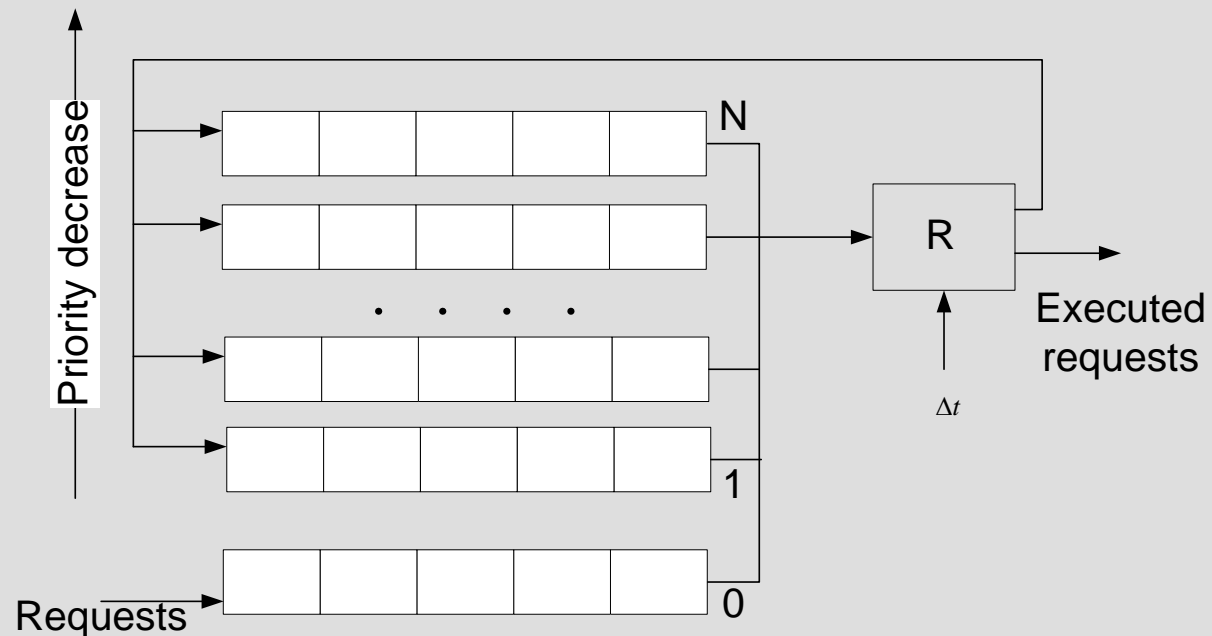
Priority scheduling

1. **Priority queues**
2. **Prioritized queues**
3. **Mixed scheduling - cyclic priority scheme**
4. **Dynamic priority scheme**

Task management disciplines

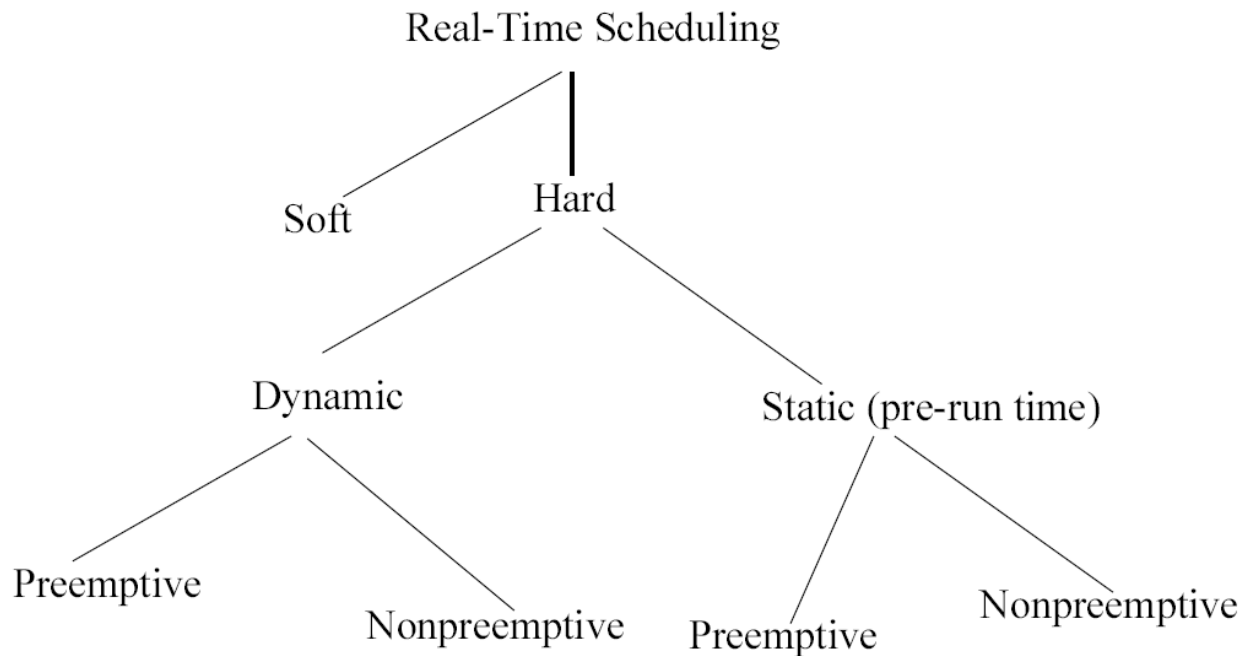
Priority scheduling

4. Dynamic priority scheme



Task management disciplines

Priority scheduling disciplines



Task management disciplines

Priority scheduling disciplines

Non-preemptive (relative) discipline

Once a task is chosen to be executed, it is run to completion even if some higher priority task becomes enabled in the meantime.

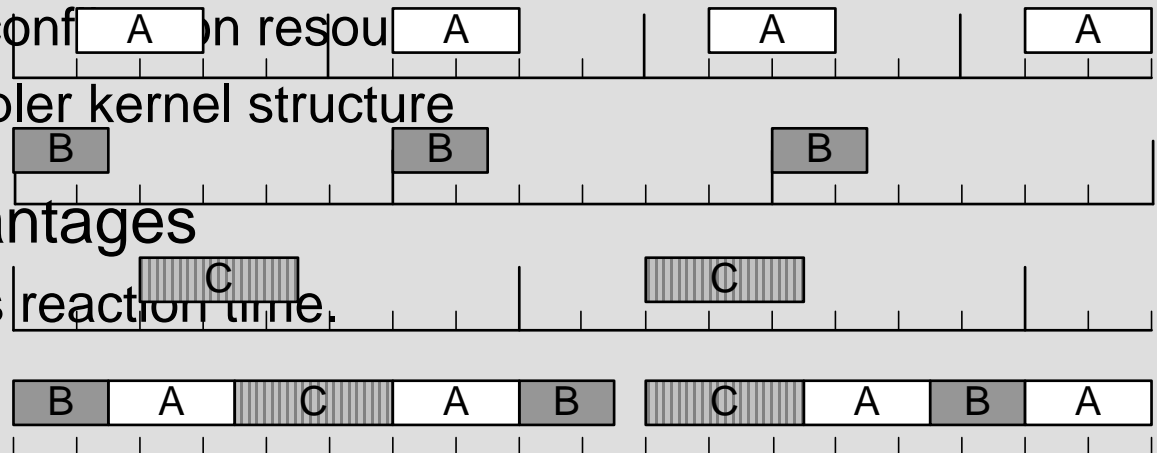
Advantages

- Better reaction
- No conflict on resources
- Simpler kernel structure

Disadvantages

- Less reaction time.

Always runs
highest priority task



Task management disciplines

Priority scheduling disciplines

Preemptive (absolute) discipline

At any time, execute the highest-priority enabled task (even if it means suspending active task).

- ◆ Advantages

- Predictability

- Simpler kernel structure



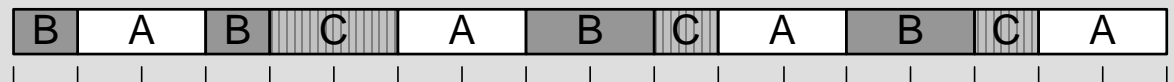
- ◆ Disadvantages

- More complicated kernel structure.

- Possible conflicts on resources.



Always runs
highest priority task
but it waits finish of
the previous task.



Kernel time-management functions

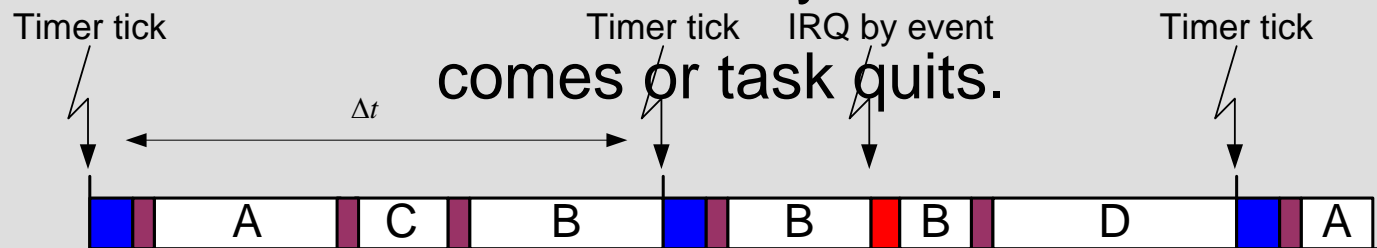
Two main types of kernels :




- ◆ Synchronous
- ◆ Asynchronous

Kernel time-management functions

Synchronous discipline

Kernel is activated only when timer tick

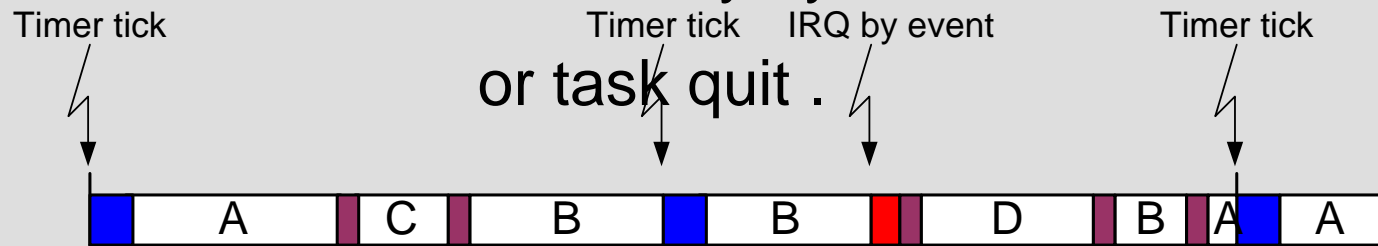



-  Timer subsystem
-  Task scheduler
-  ISR


Kernel time-management functions


Asynchronous discipline

Kernel is activated only by external event



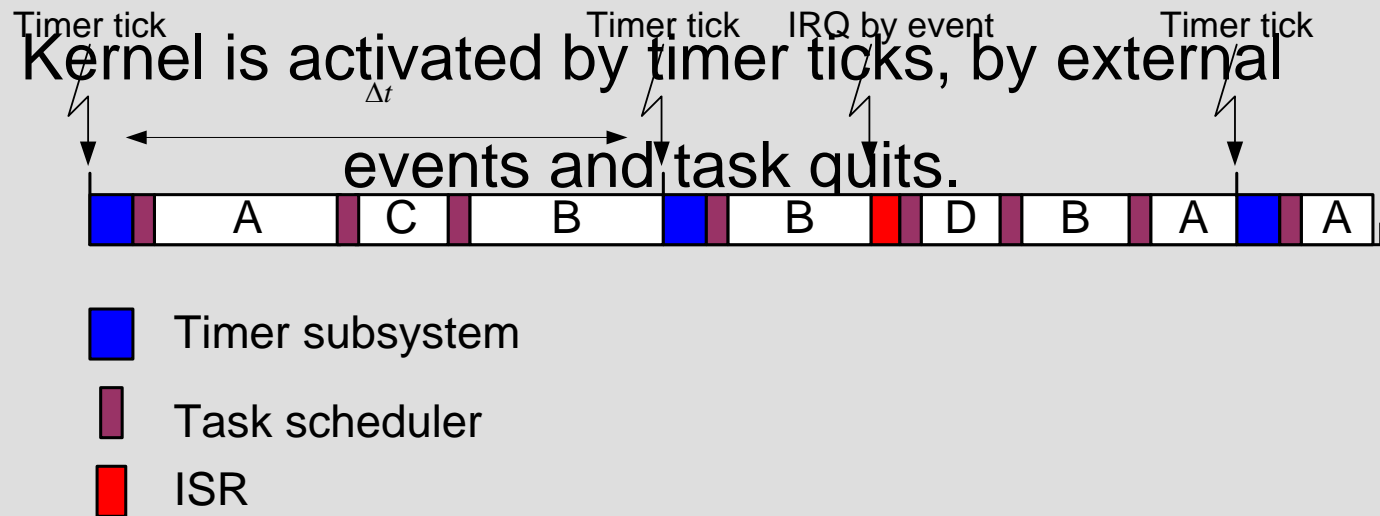
 Timer subsystem

 Task scheduler

 ISR

Kernel time-management functions

Mixed discipline (synchronous-asynchronous)



Kernel time-management functions

Functions

Time measurement subsystem is system object.

It has two main functions:

- ◆ Real-time clock
- ◆ Interval measurement.

Kernel time-management functions

Real-Time Clock

- ◆ Counts time flow.
- ◆ Activated by periodic
- ◆ Resolution _____
- ◆ Structure _____
- ◆ Absolute time counte

A number of counters:

- System tick counter
- Seconds counter
- Minutes counter
- Hours counter
- Days counter
- Months counter
- 48 ÷ 64 bits counter counting pulses generated by pulse generator

gener
to mil



Kernel time-management functions

Interval Measurement

Timer functions

Interval functions

- periodic task activation
- deadline intervals
- task budget time
- single-shot events
- blocking intervals
- delays
- priority promotion

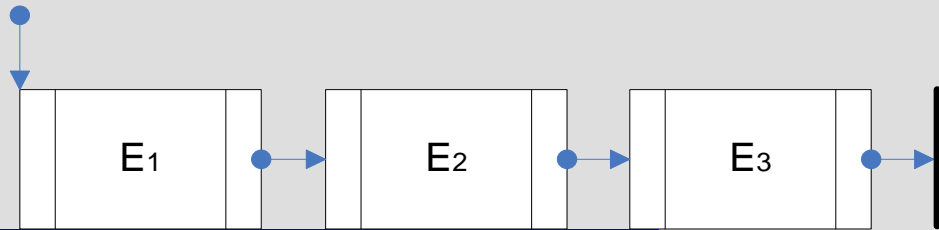
Calendar functions

- calendar-related task activation

Kernel time-management functions

Realization aspects

Timer queue:



Each element is software timer element.

Timer queue types:

- Ordered timer queue.
The first element measures the shortest interval.
- Unordered timer queue.

Kernel time-management functions

Ordered timer queue : absolute counting

Possible timer elements structure:

1. Timer queue processing is realized by comparing first timer's element **absolute time** and **absolute timer counter**. When they get equal **measured interval is finished**, timer element is removed from the queue and **associated -> task / signaling event** fulfilled.
2. If this is periodic timer new **absolute time** is calculated and timer element is included **interval** in queue again (at appropriate position).
absolute time
3. Absolute counting is more appropriate to real-time systems -> it is less time consumable and has constant execution time. Useful when intervals have to be measured.

Kernel time-management functions

Unordered timer queue : relative counting

Possible implementation structure:

1. The whole queue is processed - every counter is decremented at every timer tick. Each element which counter reaches zero is removed from the queue and associated function fulfilled.
type -> periodic / single shot
2. If this is periodic timer its **counter** is reloaded by **base value** and timer element is associated to the queue again.
associated -> task / signaling event
3. Queue processing is very time-consuming. Possible modification is to have queues for every time division measured by real-time clock.
base value
counter
4. Relative counting is very useful when calendar-associated events have to be generated – asynchronous events

External event processing functions

Event processing is usually implemented as a standard sequence of execution steps:

- ◆ Generated by means of hardware Interrupt signaling to one or more tasks about requests specific interrupt processing device event occurrence
- ◆ or Processed by Event Manager basic (non-terminal) event counting
- ◆ Event Manager consists of Event Handlers terminal event time stamping one or more tasks
- ◆ Event Handlers are activate by Interrupt Service Routines (ISR) signaling and/or invoking one or more event processing

External event processing functions

Event handler

Descriptions:

- ◆ Short executive acting as software “hook”
- ◆ Hardware-independent part of activated-by-interrupt system reactions
- ◆ Operation mode -> uninterruptible

External event processing functions

Event handler

Advantages:

- Uniform treatment of events and clear separation between standard components (event handlers) and device-specific interrupt service routines
- Transformation of physical interrupt signals into internal signals that are generated by means of vector semaphores, hence instantaneous broadcast/multicast of the event to a number of event-processing tasks
- Standard event-task interface for any type of external event and in general – for any type of event, including timing, external and internal events.
- Reduced event processing overhead resulting from the use of event counters

ISR-tasks

Main problems:

1. Interaction
2. Synchronization
3. Communication
4. Processing task association

ISR-task - Interaction

1. ***Interrupt Service Routines (ISR)*** are not schedulable part of system software.
2. They are activated by ***Interrupt Request***.
3. Depending of ***Real-Time OS*** complexity there are many different techniques to integrate ***ISR*** to the ***Real-Time OS***.
4. Priorities of ***ISRs*** are actually priorities of corresponding ***IRQs***.

ISR-task - Synchronization

ISR synchronizes to the corresponding task by:

- ◆ **Direct flags** execution requests
In a traditional environment, a variable accessible by ISR is in task scope. If ISR requests for task execution, this request does not finish by scheduler invocation. ISR has to call scheduler directly. A special kind of scheduler entry point for ISR-generated calls is needed.
- ◆ **Signaling semaphores**
Ordinary binary semaphores are used. The only difference is that call to *signal(s)* finishes without attempt to invoke system scheduler. ISR has to call scheduler directly. A special kind of scheduler entry point for ISR-generated calls is needed. Priorities of *ISRs* are actually priorities of corresponding *IRQs*.

ISR-task - Communication

Two possible solutions:

- ◆ Data transfer via common buffers – a mutual exclusion problem arises

Solutions:

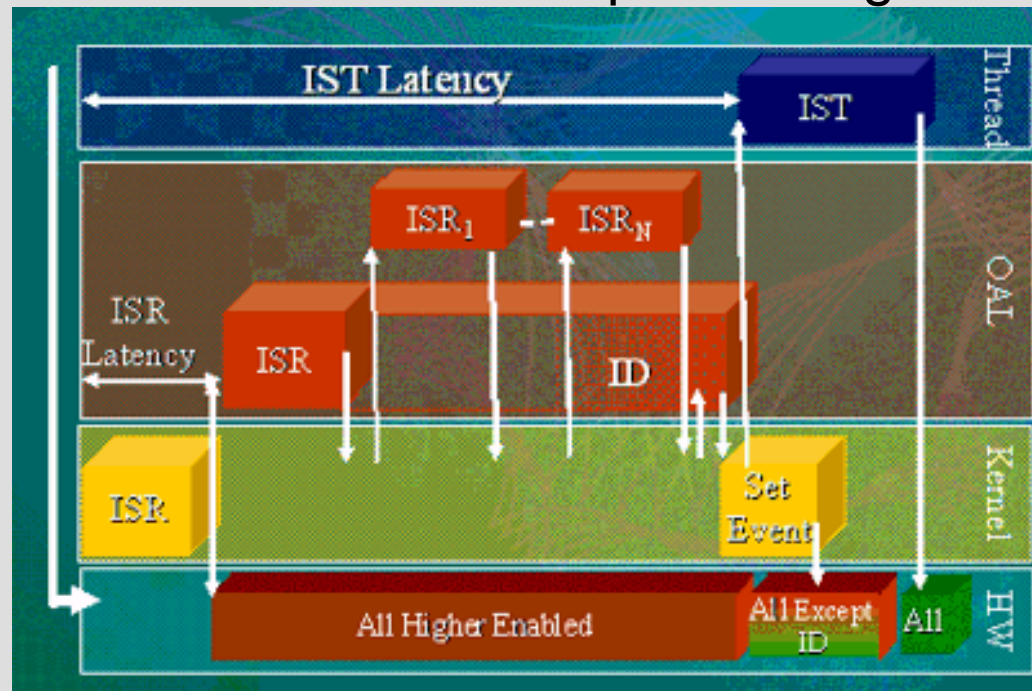
- ◆ Specially designed system primitives for ISR to process the interrupt. If access flag was set before ISR will be started, it will know that access is permitted. If flag is set, access is denied and ISR has to save its data in an alternative buffer (if needed). SET_FLAG procedure is uninterruptible region. thus ISR can be invoked *before* or *after* it but cannot interrupt it .

ISR-task - processing task association

- ISRs are associated to tasks when a task starts.
- In most cases ISR is designed together with a corresponding task. Very often they are sharing one and the same address space. This eliminates problems regarding data passing through process boundaries.
- In the context of complex Real-Time OSes for 16/32 bit processors separation between ISRs and processes is at the boundary of Hardware Adaptation Level of the kernel.

ISR-task - processing task association

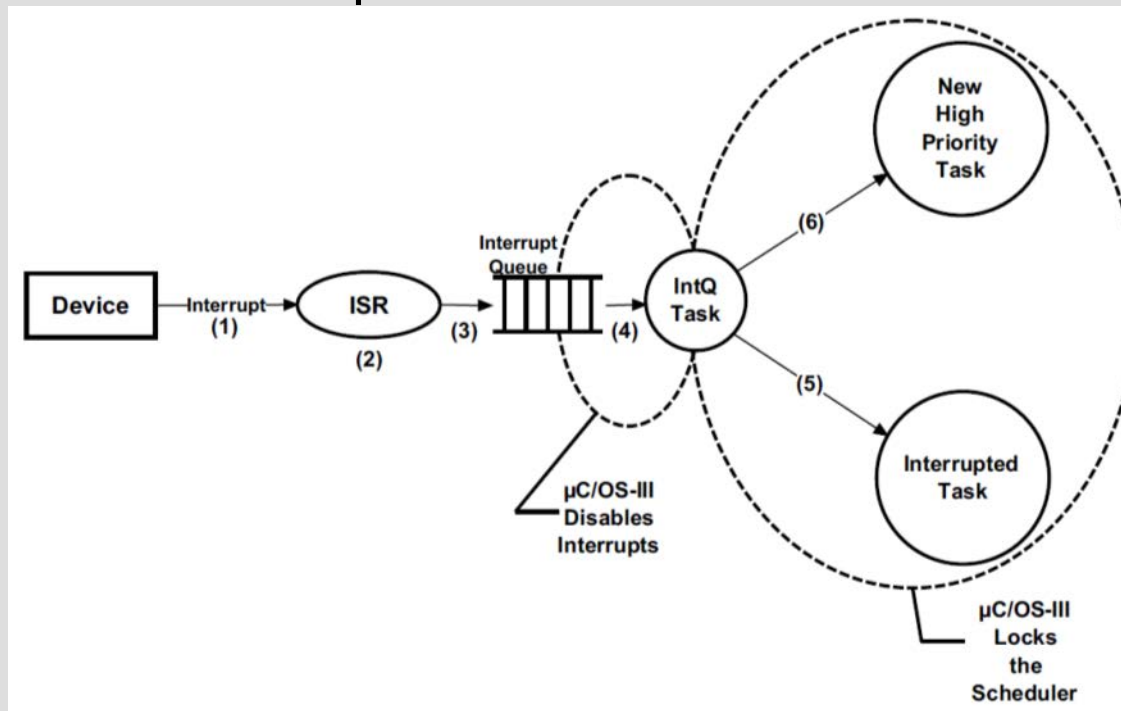
Example: WindowsCE.NET interrupt handling structure



- ◆ All OAL ISRs are processed by the IST (IST Latency). The OAL ISR (ISR) is processed by the IST (IST). The OAL ISR (ISR) is processed by the IST (IST).

ISR-task - processing task association

Example: Micrium μ C/OS-III ISR-to-task structure



Source: μ C/OS-III User's manual



Real-Time Software Environment

The End