

## Chapter 2

# BASIC CONCEPTS OF REAL TIME OPERATING SYSTEMS

Franz Rammig, Michael Ditze, Peter Janacik, Tales Heimfarth, Timo Kerstan, Simon Oberthuer and Katharina Stahl

**Abstract** Real-time applications usually are executed on top of a Real-time Operating System (RTOS). Specific scheduling algorithms can be designed. When possible, static cyclic schedules are calculated off-line. If more flexibility is needed on-line techniques are applied. These algorithms are bound to priorities which can be assigned statically or dynamically. Designing a proper RTOS architecture needs some delicate decisions. The basic services like process management, inter-process communication, interrupt handling, or process synchronization have to be provided in an efficient manner making use of a very restricted resource budget. Various techniques like library-based approaches, monolithic kernels, microkernels, or virtual machines/exokernels are applied, based on specific demands. Safety critical application can be supported by separation of applications either in the time or the space domain. Multi-core architectures need special techniques for process management, memory management, and synchronization. The upcoming Wireless Sensor Networks (WSN) generate special demands for RTOS support leading to dedicated solutions. Another special area is given by multimedia applications. Very high data rates have to be supported under (soft) RT constraints. Based on the used encoding techniques (e.g. MPEG) dedicated solutions can be created.

**Keywords:** RTOS, Scheduling, Safety Critical Systems, Wireless Sensor Networks

## 2.1 Introduction

Most embedded systems are bound to real-time constraints. In production control the various machines have to receive their orders at the right time to ensure smooth operation of a plant and to fulfill customer orders in time. Railway switching systems obviously have to act in a timely manner. In flight control systems the situation is even more restrictive. Inside technical artifacts many operations depend on timing, e.g. the control of turbines or combustion engines. This is just a small fraction of such applications. Even augmented reality systems are real-time applications as augmenting a moving reality with outdated information is useless or even dangerous.

“Real-time” means that the IT system is no longer controlling its own time domain. Now it is the progress of time of the environment which dictates how time has to progress inside the system. This environmental time may be the real one of our physical world or it may be artificially generated by some surrounding environment as well. For the embedded system there is no difference between these options. Kopetz defines real-time systems as “A real-time computer system is a computer system in which the correctness of the system behaviour depends not only on the logical results of the computation, **but also on the physical instant at which these results are produced**” [Kop97]. This means that in strict real-time systems a late result is not just late but wrong. The meaning of “late” of course has to be defined dependent on the specific application. In case of an air-bag controller it is intuitively clear what real-time means and it is easy to understand that a late firing of the air-bag is not only late but definitely wrong.

It can be concluded that in real-time systems the program logic of application tasks has to be augmented by information about timing. Such timing information contains the earliest point of time the task may be started as well as the latest allowed finishing time. This, together with the program logic may be seen as a specification for the computing system what to do and when to do it.

Many such tasks may have to be executed concurrently on an embedded computing system. Such situations usually are handled by some kind of operating system. The same is true in case of real-time systems. But now an additional objective function is introduced, an objective function which dominates most other ones: Formulated real-time constraints have to be respected. An operating system which is capable of taking care of this is called a “Real-time Operating System (RTOS)”. Of course some additional information is needed by an RTOS to manage real-time tasks. Especially the worst-case execution time (WCET) on the specific target architecture of any real-time task has to be available. Determining the WCET of a task is a demanding goal on its own. It must never be underestimated. On the other hand the potential

over-estimation has to be reduced as far as possible to allow efficient system implementations.

The above discussion indicates that we first have to discuss fundamental properties of real-time tasks. On this basis we can then introduce basic techniques used in RTOS to handle such tasks. We will concentrate on real-time scheduling and on schedulability analysis.

## 2.2 Characteristics of Real-Time Tasks

First of all a real-time task is a task like any other. However, there is an essential difference to other computation: the notion of *time*. *Time* means that the correctness of the system depends not only on logical results but also on the time the results are produced. In contrary to other classes of systems in a real-time system the system time (*internal time*) has to be measured with same time scale as the controlled environment (*external time*). One parameter constitutes the main difference between *real time* and *non-real-time*: the **deadline**. Any postulated deadline has to be met under all (even the worst) circumstances. This has the consequence that real-time means predictability. It is a widespread myth that real-time systems have to be fast. Of course they have to be fast enough to enable guaranteeing the required deadlines. Most of all, however, a real-time system has to be predictable. Ensuring this predictability even may slow down a system.

Real-time systems can be characterized by the strictness of real-time restrictions.

A real-time task is called *hard* if missing its deadline may cause catastrophic consequences on the environment under control. Typical application areas can be found in the automotive domain when looking at e.g. power-train control, air-bag control, steer by wire, and brake by wire. In the aeronautics domain engine control or aerodynamic control may serve as examples.

A RT task is called *firm* if missing its deadline makes the result useless, but missing does not cause serious damage. Typical application areas are weather forecast or decisions on stock exchange orders.

A RT task is called *soft* if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage. Here typical application areas are communication systems (voice over IP), any kind of user interaction, or comfort electronics (most body electronics in cars).

Concerning timing, a real-time task  $J_i$  can be characterized by the following parameters: Arrival time  $a_i$ , WCET  $C_i$ , (absolute / relative) deadline  $d_i / D_i$ , start time  $s_i$ , finishing time  $f_i$  (see Fig. 2.1).

The *arrival time*  $a_i$  is the time  $J_i$  becomes ready for execution. Sometimes it is also called *request time* or *release time*, denoted by  $r_i$ . It is a parameter

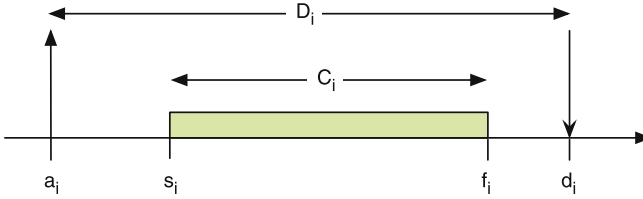


Figure 2.1. Parameters of a real-time task.

under control of the application task. A task which is not known to the RTOS obviously is also not considered by it.

Another parameter that comes with the application task is its *computation time*  $C_i$ . This is the WCET which has to be determined previously and has to be known by the RTOS. Of course it is the WCET only under the assumption that the task is not interfered by any other task. Interference can happen only when managed by the RTOS. So any influence by interference due to other tasks is known by the RTOS and has to be considered by the RTOS.

The third parameter that comes with the application task is its *deadline*. Here a distinction has to be made between an *absolute deadline*, denoted by  $d_i$  and a *relative* one, denoted by  $D_i$ . Absolute deadline means a value with respect to the global time of the entire system while relative deadline means relative to the arrival time of the respective task. In any case, it has to be guaranteed by the RTOS that the tasks will be finished not later than the deadline, independent from any circumstances, even the worst imaginable ones.

The remaining two parameters are under control of the RTOS. It is the RTOS that decides when to start an application task, i.e. to set the *start time*  $s_i$ . Of course it can never be earlier than the arrival time  $a_i$  as before this time the task is entirely unknown to the RTOS.

In a similar manner it is up to the RTOS when a task reaches its *finishing time*  $f_i$ . It can be calculated to be at least  $s_i + C_i$ . However,  $f_i$  may be disturbed by other tasks so that the finishing time  $f_i$  may be later. Whenever this happens, it happens under control of the RTOS. Therefore it is the responsibility of the RTOS to guarantee that  $f_i$  is not later than the respective deadline.

Orthogonally to the distinction into soft, firm, and hard real-time two main classes of tasks can be identified: *periodic* and *aperiodic* ones. Both types are generic tasks, i.e. over time a sequence of instances is generated. Usually such an instance is called *job*. All instances share the same code and therefore the same WCET and relative deadline. In case of periodic tasks these instances show up with a fixed *period*, denoted by  $T_i$ . This means that once knowing the first arrival time, all following arrival times are pre-defined. The first arrival

time, i.e. the arrival time of the first instance usually is called the phase of this (generic) task, denoted by  $\phi_i$ .

In the case of aperiodic tasks no period is present, i.e. the next arrival time of an instance of an aperiodic task is unknown a priori and may happen at any time. Usually the assumption is made, that up to the absolute deadline of a task instance no additional instance will be issued into the system.

Periodic tasks reflect directly the “sense–execute–act” loop in control applications. They therefore represent the main workload of embedded systems. Any RTOS usually is optimized into the direction of handling such tasks in an optimized manner. Aperiodic tasks appear for initialization reasons, for setting of parameters and, most importantly, for the handling of interrupts that show up in an aperiodic manner. There is a certain style of programming embedded systems which reduces the software to a strictly event driven system. The NesC programming language used for TinyOS [CHB<sup>+</sup>01] follows this principle. So, whenever this style of programming has to be supported, the handling of aperiodic tasks becomes a major issue of the RTOS.

Tasks of a given task set may be independent or dependent. A task  $J_i$  is called dependent on task  $J_k$  if  $J_i$  cannot be started before  $J_k$  has been finished. Dependence is a transitive property. A task  $J_i$  is called direct predecessor of task  $J_k$  if there is no task  $J_m$  between them such that  $J_m$  is dependent on  $J_i$  and  $J_k$  is dependent of  $J_m$ . Dependencies can be defined using a directed acyclic graph (DAG). Obviously dependencies introduce additional constraints that need to be handled by the RTOS.

Unfortunately direct support for expressing dependencies is rarely found in modeling and programming languages. UML Sequence Charts represent dependencies, however in a rather unwieldy manner. In programming languages dependencies have to be coded in detail and therefore are hard to be identified in the program code.

Another constraint on task sets is introduced by non sharable resources. A *resource* is any object to be used by a task. In HW this may be some circuitry like an ALU or a bus, in SW it may be a certain data structure, a set of variables, or a memory area. A resource is called *private resource* if it is dedicated to a particular task, i.e. it is not used by any other one. In contrary to this a *shared resource* is to be used by more than one task. In HW a bus is a typical example of a shared resource. It is also a typical example for this class of shared resources that need most care in handling: an *exclusive resource* is a shared resource where simultaneous access from different tasks is not allowed. Coming back to the bus example, a bus is an ordinary shared resource from the point of view of components reading from this bus but an exclusive one for any writer.

Like in the case of dependencies, direct support for specifying the class of resources is lacking in both, modeling and most programming languages.

There are techniques to handle such cases. In HW design special arbiters have to be included into the circuit. Unfortunately in VHDL, e.g., they are just components like any others, i.e. they cannot be identified easily. In SW the concept of a so called *critical section* is introduced. This is a piece of code that is to be executed under mutual exclusion constraints. The management then has to be coded directly, e.g. using semaphores [Dij68]. They constitute the link to the operating system as it is the OS which provides the semaphore operations as system services. It will be shown later that the concept of semaphores needs careful rethinking when real-time systems have to be built.

To sum up, real-time tasks can be characterized by a well defined set of parameters. Fortunately in most publications the same abbreviations are used for them.

$\Gamma$  set of tasks. This set may consist of aperiodic ones, periodic ones, or both.

$\tau_i$  a generic task. This means that over time many instances of this task will exist.

$\tau_{i,j}$  instance  $j$  of task  $\tau_i$ .

$r_{i,j}$  *release time* of  $\tau_{i,j}$ . The release time is an absolute value and specific for each instance.

$\phi_i$  phase of  $\tau_i$  ( $= \tau_{i,1}$ , i.e. release time of first instance). It is a parameter of the entire generic task.

$T_i$  period of  $\tau_i$  ( $=$  interval between two consecutive activations).

$D_i$  relative deadline of  $\tau_i$  (relative to release time, therefore a parameter of the entire generic task).

$d_{i,j}$  absolute deadline of  $\tau_{i,j}$  ( $d_{i,j} = \phi_i + (j - 1)T_i + D_i$ ). It is a property of the specific instance.

$s_{i,j}$  start time of  $\tau_{i,j}$  ( $s_{i,j} \geq r_{i,j}$ ). It is an absolute value and specific for each instance.

$f_{i,j}$  finishing time of  $\tau_{i,j}$  ( $f_{i,j} \leq d_{i,j}$ ). It is an absolute value and specific for each instance.

### 2.3 Real-Time Scheduling

Given is a set of  $n$  generic tasks  $\Gamma = \{\tau_1, \dots, \tau_n\}$ , a set of  $m$  processors  $P = \{P_1, \dots, P_m\}$ , and a set of  $s$  resources  $R = \{R_1, \dots, R_s\}$ . There may exist precedences, specified using a precedence graph (DAG) and, as we are considering real-time systems, timing constraints are associated to each task.

The goal of *real-time scheduling* is to assign processors from  $P$  and resources from  $R$  to tasks from  $\Gamma$  in such a way that all task instances are completed under the imposed constraints. This problem in its general form is NP-complete! Therefore relaxed situations have to be enforced and/or proper heuristics have to be applied. In principle scheduling is an on-line algorithm. Under certain assumptions large parts of scheduling can be done off-line. Generating static cyclic schedules may serve as an example. In any case all exact or heuristic algorithms should have very low complexity.

In principle scheduling algorithms may be *preemptive* or *non preemptive*. In preemptive approaches a running task instance may be stopped (preempted) at any time and restarted (resumed) at any time later. Any preemption means some delay in executing the task instance, a delay which the RTOS has to take care of as it has to guarantee respecting the deadline. In case of non preemptive scheduling a task instance once started will execute undisturbed until it finishes or is blocked due to an attempt to access an unavailable exclusive resource. Non preemptive approaches result in less context switches (replacement of one task by another one, usually a very costly operation as many processor locations have to be saved and restored). This may lead to the conclusion that non preemptive approaches should be preferable in real-time scheduling. However, not allowing preemption imposes such hard restrictions on the scheduler's freedom that for most non-static cases predictable real-time scheduling solutions with an acceptable processor utilization rate are known only if preemption is allowed. In the sequel basic preemptive real-time scheduling algorithms for periodic and aperiodic tasks will be discussed shortly.

### 2.3.1 Rate Monotonic Priority Assignment

All real-time scheduling algorithms strictly rely on priorities. So the basic principle is that at any point of time always this task instance  $\tau_{ij}$  is executed which has the highest priority among all active task instances. A task instance  $\tau_{ij}$  is active in the period between its release time  $r_{i,j}$  and its finishing time  $f_{i,j}$ . In this section it is assumed that a task set  $\Gamma$  of independent tasks  $\tau_i$  with no resource conflicts has to be scheduled.

*Rate Monotonic Priority Assignment (RM)* is a so-called static priority scheduling algorithm. In such algorithms priorities are assigned a priori and are never modified during runtime of the system. RM assigns priorities simply in accordance with its periods, i.e. the priority is as higher as shorter is the period which means as higher is the activation rate. So RM is a scheduling algorithm for periodic task sets. It is assumed that the periods of the different tasks differ, we have so called *multi-rate* systems (handling of single-rate systems is trivial). In addition it is assumed that the relative deadlines of the tasks are identical to the periods ( $D_i = T_i$ ). RM is intrinsically preemptive as it may happen that

a task instance is running when a new instance of a lower-period, i.e. higher priority task is released. In such a case the currently running task is preempted in favor of the newly arriving one. See Fig. 2.2 for an example of a schedule produced by RM. There are two tasks,  $\tau_1$  (period  $T_1 = 3$ , WCET  $C_1 = 1$ ) and  $\tau_2$  (period  $T_2 = 7$ , WCET  $C_2 = 4$ ). The first two instances of  $\tau_2$  are preempted once, the third one twice due to starting new instances of the higher priority  $\tau_1$ .

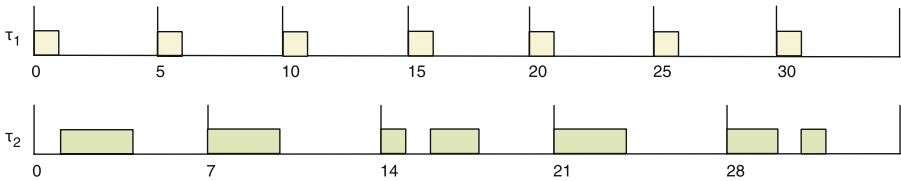


Figure 2.2. RM example schedule.

It can be shown [But04, p. 78ff] that RM is optimal among all fixed priority scheduling algorithms in the sense that if RM does not provide a feasible schedule than no other fixed priority algorithm can.

A hard real-time system cannot be started before carefully analyzing its schedulability. A specific schedule is called *feasible* if all instances  $\tau_{ij}$  of all tasks  $\tau_i$  can be completed according to a set of specified constraints. A set of tasks is called *schedulable* if at least one algorithm does exist that can produce a feasible schedule. When applied to RM the algorithm has already been selected. The question now is to decide a priori whether a given task set  $\Gamma$  is schedulable by RM.

A simple test is given by comparing the utilization of the given task set with the utilization of the worst imaginable task set which is still schedulable by RM. This constitutes a least upper bound (LUB) of utilization among all potential task sets.

Given a set  $\Gamma$  of aperiodic tasks the *processor utilization factor*  $U$  is the fraction of processor time spent in the execution of the tasks set.  $C_i/T_i$  is the fraction of processor time spent in executing task  $\tau_i$ . The utilization  $U$  of  $\Gamma$  can be calculated simply by the sum

$$U = \sum_{i=1}^n \frac{C_i}{T_i}. \quad (2.1)$$

Obviously this can be done off-line as no runtime parameters are used in this formula. It can be shown [But04, p. 87ff] that the utilization LUB,  $U_{\text{lub}}$ , i.e. the utilization of the worst case task set is given by  $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$  which converges towards  $U_{\text{lub}} = \ln 2 \approx 0.69$  with increasing  $n$ . As the number  $n$  of tasks in the given task set  $\Gamma$  is known a priori as well,  $U_{\text{lub}}$  can be calculated



off-line and as a consequence the schedulability analysis can be performed off-line. Unfortunately  $U_{\text{lub}} = n(2^{\frac{1}{n}} - 1)$  is sufficient but not necessary to guarantee the feasibility of a given task set. There may exist specific task sets which are schedulable under RM despite the fact that for their utilization  $U$  it holds that  $U_{\text{lub}} < U < 1$ .

$$\sum_{i=1}^n \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1), \quad (2.2)$$

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_j}{T_j} \right\rceil C_j. \quad (2.3)$$

In RM the assumption is made that the relative deadlines of the tasks are identical to the periods ( $D_i = T_i$ ). This restriction can be relaxed easily by replacing  $T_i$  by  $D_i$  in the definition of priority assignment. The algorithm then is called *Deadline Monotonic Priority Assignment (DM)*. Even the schedulability analysis can be transferred directly resulting in the condition as in Eq. 2.2, which in this case, however is even more pessimistic than in the RM case. A crisp schedulability test for fixed priority assignment strategies like RM and DM is given by the so-called *Response Time Analysis*. In this case for each task  $\tau_i$  the largest finishing time among all instances  $\tau_{ij}$  with respect to its relative deadline  $D_i$  is calculated precisely. This largest finishing time is called *response time*  $R_i$  of task  $\tau_i$ . If for all tasks  $\tau_i$  of the given task set  $\Gamma$   $R_i$  is not greater than the relative deadline  $D_i$ , schedulability is proven.

The response time  $R_i$  can be calculated by  $R_i = C_i + I_i$  where  $C_i$  is the WCET of  $\tau_i$  and  $I_i$  is the interference due to pre-emption by higher priority tasks. The question is how to calculate  $I_i$ . For this we have to sum up over all higher priority tasks  $\tau_j$ ,  $j < i$  the number of interferences given by  $\lceil R_i / T_j \rceil$  multiplied by the duration of the respective interference  $C_j$ . This results in the definition of the response time  $R_i$  of task  $\tau_i$  as shown in Eq. 2.3.

Unfortunately this is a recurrent equation as the argument  $R_i$  stands on both sides of the equation. By an iterative algorithm, however we can calculate the least fixpoint of the equation. If it is less or equal to the relative deadline the test for this specific task is successful, otherwise it fails. The test is successful for the entire task set  $\Gamma$  if it does not fail for a single task  $\tau_i$ . So this test is rather computation intense. Fortunately it can be carried out off-line as no runtime parameters have to be known.

### 2.3.2 Earliest Deadline First Scheduling

In contrary to RM or DM, Earliest Deadline First (EDF) scheduling is a dynamic priority assignment. Now task instances  $\tau_{ij}$  always get assigned a

priority inverse proportional to their absolute deadline  $d_{ij}$  i.e. the priority is as higher as the absolute deadline is shorter (ties are broken in favor of already running task instances). This means that whenever a task instance is released the priorities have to be re-calculated and the priority of a task (i.e. of its instances) may vary during runtime. Despite this difference the handling of task instances is the same as in the case of RM or DM: At each instance of time this task instance is executed that currently has the highest priority among all active task instances. Therefore, like RM or DM, EDF is intrinsically preemptive. Figure 2.3 shows an example schedule produced by EDF for the same task set as used in Fig. 2.2. The third instance of  $\tau_2$  is preempted only once as in the case of equal absolute deadlines the already running task is preferred.

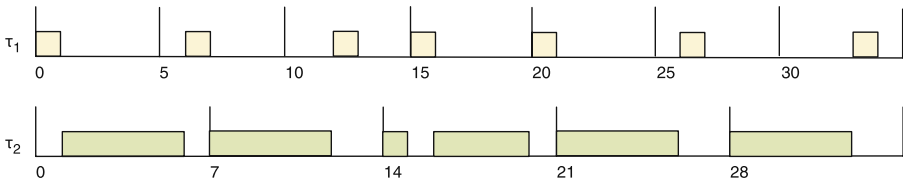


Figure 2.3. EDF example schedule.

It can be shown [But04, p. 51ff, 92] that EDF is optimal among all periodic task scheduling techniques in the sense that if EDF does not provide a feasible schedule then no other periodic task scheduling algorithm can. Another good property of EDF is that schedulability analysis is really simple for EDF. A simple utilization test can be applied where  $U_{\text{lub}} = 1$ , i.e. the utilization just has to be compared with the constant 1.

EDF can also be applied to aperiodic task sets. Its optimality guarantees that the maximal lateness is minimized when EDF is applied. Lateness  $L_{i,j}$  of a task instance  $\tau_{ij}$  is defined as the time between absolute deadline and finishing time:  $L_{i,j} = f_{i,j} - d_{i,j}$ .

So it seems that EDF has only advantages over fixed priority algorithms. Despite this fact those algorithms still serve as the workhorse in most RTOS systems. It is argued that EDF is more complicated to implement as at runtime it has to rearrange priorities while RM or DM do not. EDF is also considered to be extremely sensitive to overload conditions where a so-called *Domino Effect* may happen, i.e. missing a single deadline may result in missing the deadlines of all tasks of a task set. In a recent publication [But05] however, it has been shown that most of the arguments against EDF are not relevant in practical applications.

## 2.4 Operating System Designs

The most common Operating Systems are based on kernel designs. The kernel design has been around for almost 40 years and offers a clear separation between the operating system and the application running on top of it, as they are allocated in different memory locations. The processes can use the kernel functionality by performing system calls. System calls are software interrupts which allow switching from the application to the operating system. Therefore the kernel needs to install an interrupt handler for different modes of operation, depicted in Fig. 2.4, that can be enabled in the program status word (PSW): User mode and Supervisor mode. For this reason, protection is done in modern SoCs at peripheral side. Some registers can be changed only if the CPU signals a specific execution mode (e.g. master mode) via a set of additional HW-signals in the bus infrastructure.

Processes outside the OS are executed within user mode and are not allowed to execute instructions which are only available in supervisor mode. This means that the user mode instructions constitute a non-critical subset of the supervisor mode instructions. During runtime of a process the supervisor mode bit within the PSW is disabled and can only be enabled if an interrupt such as a system call or an external interrupt occurs. The operating system is responsible for enabling the user mode at the time a user process is activated. Typically a user process has its own virtual memory address space which separates it completely from the kernel. However this is not possible on all embedded microcontrollers as they may lack a memory management unit (MMU) enabling the use of virtual memory.

The use of virtual memory, if there is a MMU available, has to be realized without any unbound memory accesses like swapping on an external disk or replacing translation lookaside buffer (TLB) entries by searching a dynamically sized page table.

To use the functionality provided by the OS kernel it is necessary to define an interface that allows applications to use it. This interface is called the appli-

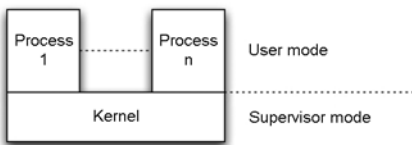


Figure 2.4. Execution modes.

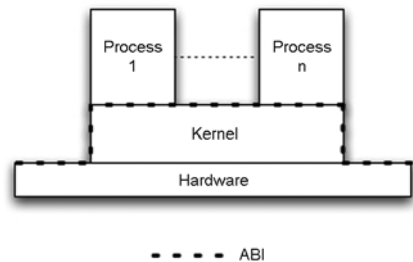


Figure 2.5. Application binary interface.

cation binary interface (ABI). The ABI defines a set of **system calls**, a **register usage convention**, a **stack layout** and enables binary compatibility whereas an application programming interface (API) enables source code compatibility through the definition of a set of function signatures providing a fixed interface to call these functions. Figure 2.5 shows the location of the ABI within an architectural schemata.

The kernel itself can be built in many ways and usually provides the following basic activities: **Process management**, **process communication**, **interrupt handling**, and **process synchronization**.

Process management is responsible for process creation, process termination, scheduling, dispatching, context switching and other related activities.

Interrupt handling in a RTOS is different to the standard implementation of an ordinary OS. In an ordinary OS interrupts can preempt running processes at any time. This can lead to unbound delays which are not acceptable in a RTOS. Therefore the handling of interrupts is integrated into the scheduling so that it can be scheduled along with the other processes and a guarantee of feasibility can be achieved even in the presence of interrupt requests.

Another important role of the kernel is to provide functionalities for the synchronization and communication of processes. The use of ordinary semaphores is not possible within a RTOS as the caller may experience unbound delays in case of a priority inversion problem. Therefore the synchronization mechanisms need to support a resource access protocol such as Priority Inheritance, Priority Ceiling or Stack Resource policy [But04, p. 191ff].

As already stated there are different ways to realize a kernel. Today the main design question is whether to use a monolithic kernel, a microkernel or a combination called hybrid kernel [Sta01, Tan01].

### 2.4.1 Library-Based RTOS (“Kernel-Less” Approach)

For systems without MMUs the RTOS can be built as a library which is linked together with the application. This results in one single executable which is executed in one single address space. Therefore no loader is required to dynamically load applications at run-time, by this minimizing the operating system code. Another advantage of a library-based RTOS and the execution in a single common address space is that system calls can be simply implemented as function calls. Thus no context-switches are required when calling an operating system function. This is often more efficient and less time consuming as a full context switch with address space changes when having an RTOS implemented as a kernel in a separated address space. The disadvantages of a library based RTOS running on systems with no “full MMU” is the lack of security through hardware memory separation. All application and operating system activities have to be implemented as threads in the same address space.

Bugs in one part of the system can easily affect the whole system. But on small microcontrollers on which only one application is executed this disadvantage is acceptable.

An example for a library based operating system is the operating system library DREAMS. Operating systems and run-time platforms for even heterogeneous processor architectures can be constructed from customizable components skeletons out of the DREAMS (**D**istributed **R**ea-time **E**xtensible **A**pplication **M**anagement **S**ystem) library [Dit99]. By creating a configuration description all desired objects of the system have to be interconnected and customized afterwards in a fine-grained manner. The primary goal of that process is to add only those components and properties that are really required by the application.

### 2.4.2 Monolithic Kernels

The monolithic approach of building a kernel is straightforward. All functionality provided by the OS is realized within the kernel itself. “The structure is that there is no structure” [Tan01]. The kernel consists of a set of procedures which are able to call each other without any restrictions. Figure 2.6 shows a call graph of a totally unstructured monolithic kernel versus a monolithic kernel which is separated into service functions and help functions to bring at least some structure into the kernel. The service functions are the entry points for the interrupts which are demultiplexed in the main function and delegated to the associated service function.

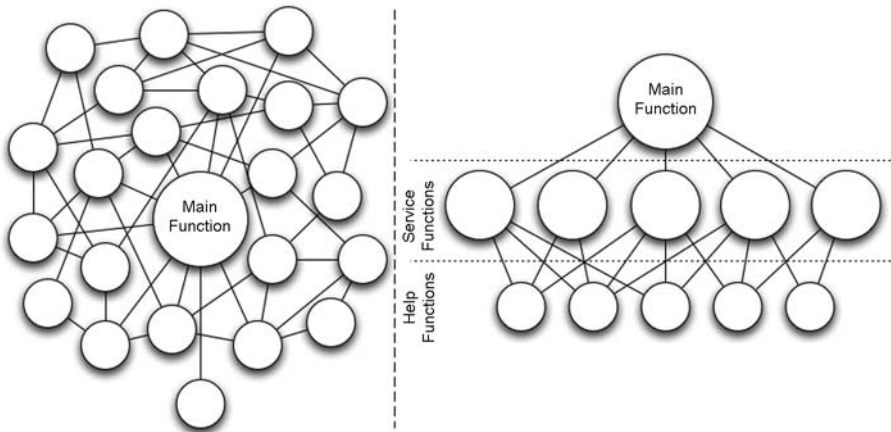


Figure 2.6. Unstructured vs. structured monolithic kernel.

The service functions can use any support function they need. The main advantage of monolithic kernels is their performance. As reaction to a sys-

tem call a context switch to the operating system has to be performed and the appropriate service functions have to be executed in the kernel space. This is pretty simple, as there are only function calls that need to be performed.

In the case of monolithic kernels it cannot be excluded that any single fault occurring within the kernel functions can lead to a total crash of the whole system. In most cases device drivers included in a monolithic kernel are very error-prone. Several studies on software dependability report fault densities of 2 to 75 bugs per 1000 lines of executable code. Drivers, which typically comprise 70% of the operating system code, have a reported error rate that is 3 to 7 times higher. A common example that can lead to a total crash is an unchecked pointer that may contain a wrong address. This results in overwriting of sensitive kernel data such as the kernel code itself [OW02, Sta01, Tan01, BP84, THB06].

### 2.4.3 Microkernels

To clean up the structural mess of monolithic kernels Fig. 2.7 shows microkernel design was developed. It reduces the services provided by the kernel dramatically by putting all services, which are not essentially necessary for the microkernel, into user space as isolated processes. The service processes typically behave like servers of the client-server model. To use such a service an application needs to send a message with a service request to the service which receives the request, completes the request and sends back a response message to the client application.

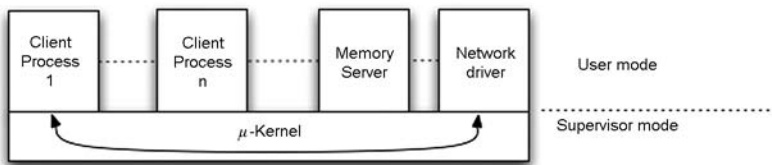


Figure 2.7. Microkernel architecture.

The big question is which services are not essentially necessary for the microkernel. The common approach puts the following services into the microkernel itself: **Dispatcher**, **Scheduler**, and **Memory Manager**.

Whether it is necessary to put the memory manager into the microkernel is a topic that has been discussed for a long time without any general agreement. However some memory management for the kernel objects itself is needed within the microkernel.

The big advantage of microkernels against monolithic kernels is the clear separation of services from the kernel itself making the kernel a very small

piece of software that provides a better fault isolation and can be maintained more easily than a monolithic kernel. The fault isolation prevents crashing the whole system. Even if e.g. a driver located in user space fails it is not possible for the driver to manipulate any kernel sensitive data like the kernel code.

The price we have to pay for the better structuring and fault isolation is that we get a high amount of interprocess communication through message passing and a high amount of context switching. The reason for this is that for every system call at least two messages have to be sent and four context switches have to be performed. This is illustrated in Fig. 2.8.

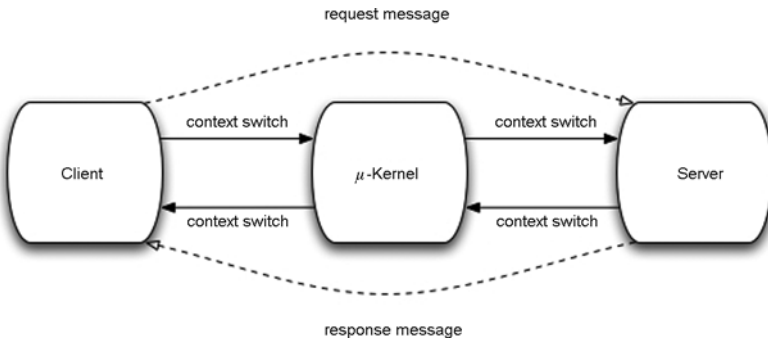


Figure 2.8. Client/Server IPC.

In contrast to monolithic kernels we also have to deal with an impact on the real-time behavior, because now system calls are not necessarily executed at the time they have been initiated. The reason is that the services behave like regular processes that have to be scheduled by the real-time scheduler. There are several approaches to deal with that problem. A very simple one is to use priority message queues for the service requests within the server and to apply priority inheritance on the server processes to guarantee that no unbound blocking time can occur [Sta01, Tan01].

#### 2.4.4 Virtual Machines and Exokernels

The main idea of system virtual machines is to provide an exact copy of the available hardware for every virtual machine. Therefore a small control program is necessary to assign the available hardware to the virtual machines. This program is called the virtual machine monitor (VMM) or hypervisor (cf. Fig. 2.9). This program is the only code executed in supervisor mode and ensures that the virtual machines are clearly isolated from each other.

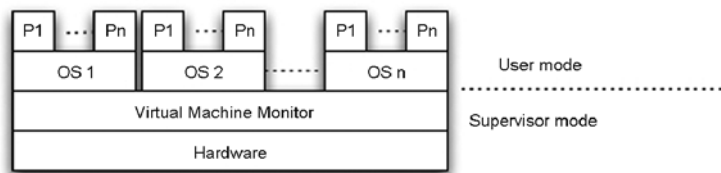


Figure 2.9. Virtual Machine Monitor.

The biggest issue to be solved is the question whether virtualization can be achieved efficiently. To answer this the instruction set architecture (ISA) plays the most important role. The ISA is divided into sensitive and innocuous instruction. Sensitive instructions interact with hardware and need to cause a trap to activate the VMM. Innocuous instruction can be executed natively if possible (provided that the ISAs of the host and the virtual machine are identical). If instructions cannot be executed natively they need to be emulated. For emulation the target code to be executed on a different host ISA needs to be transformed before it can be executed. The question whether an efficient VMM can be built is reduced to the question whether the set of sensitive instructions is a subset of the set of privileged instructions as in Fig. 2.10 [PG74].

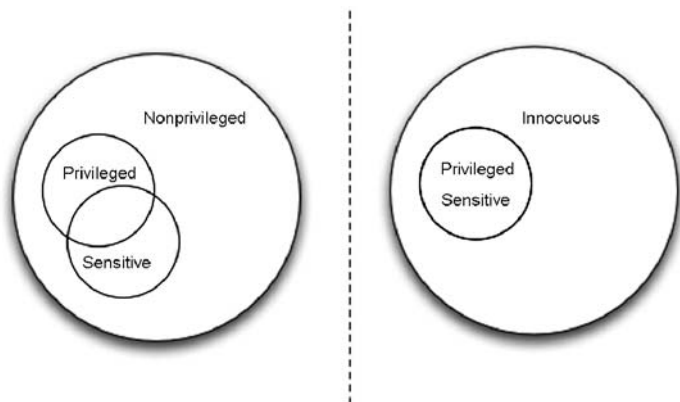


Figure 2.10. Efficiency classification of ISAs.

Exokernels are very similar to virtual machine monitors, but they differ in the way that exokernels do not provide an exact copy of the available hardware. Instead they partition the available resources and assign them to the virtual machines running on top of the exokernel. A good example is the main memory of the system. VMMs provide an exact copy of the complete main memory



to the virtual machines running on top of the VMM. The VMM needs to map the memory of every virtual machine to the real physical memory. Exokernels do not have to manage such a mapping as the different virtual machines would have only access to disjoint subsets of the available physical memory [Tan01, SN05].

## **2.5 RTOS for Safety Critical Systems**

Computer systems that operate systems of critical responsibility are called safety-critical systems. Typically, a small deviation in the environment or the system's behavior, a failure or an error appearing within such a system can yield in hazardous situations and may cause catastrophes. Safety-critical systems therefore must not only guarantee real-time behaviour but furthermore they require absolute dependability and availability of system service. To free application developers from implementing safety and real-time mechanisms into each application, operating systems serve as the underlying platform designed towards supporting real-time and all safety-incorporating non-functional features.

Because of the critical consequences of a system failure, standards are required to specify the design and the development process. They define the methods and techniques that are required to prevent system failures and enforce a state-of-the-art quality-of-service in safety-critical applications. Two relevant standards exist: IEC 61508 and DO-178B. The title of the international standard IEC 61508 is "Functional safety of electrical/electronic/programmable electronic safety-related systems". It is a generic safety standard that forms the basis for many other—domain specific—standards. This standard defines requirements on the lifecycle of safety-related systems, from system development to its operation. It identifies measures and techniques for preventing failures and contains methods for controlling possible system failures. DO-178B is titled "Software Considerations in Airborne Systems and Equipment Certification" and specifies guidelines for the development of avionic software. It builds up a stringent application-dependent safety standard.

As recent trends are heading towards the integration of applications of different criticality levels on one single platform, operating systems for safety-critical applications face the challenge of guaranteeing the availability of the processor time as well as the availability of resources (full protection in time and in space domain). These challenges must be inherently incorporated into the RTOS architecture. The Avionics industry formulated these architectural requirements in the ARINC 653 specification to guide manufacturers of avionic application software towards maximum standardization.

### 2.5.1 Protection in Time Domain

Running multiple applications with different criticality levels on one processor may lead to no provision for guaranteeing processor time for critical applications. Consider the following scenario: Two applications of different criticality levels, each with one thread at the same priority run on a single system. Thread 1 is a non-critical thread whereas thread 2 is a critical one that needs at least 45% of the processor time to process its workload. As the two threads get assigned the same priority, a scheduler will assign each of the threads 50% of the processor time. In that case, the critical thread 2 will get its work done. Suppose that thread 1 spawns a new thread with the same priority. Then, the scheduler handles three threads at the same priority. As a consequence each of the threads will get only 33% of the processor time. Hence, the critical thread 2 is not able to handle its workload any more. The requirement of protection in time domain results clearly from this example.

### 2.5.2 Protection in Space Domain

Due to predictability reasons, many RTOS designers do not use virtual memory management. The fact that multiple applications with different criticality levels run on one single processor involves that processes share the same memory space. This implies that a process is able to corrupt the code, data or the stack of another process, intentionally or unintentionally. Furthermore, a process can also corrupt data or code of the operating system kernel which affects the safety and reliability of the system. In fact, it can lead to unexpected system behavior that infects the predictability and it can even bring down the entire system. Therefore, the protection of the memory is one key issue in RTOS for safety critical systems.

### 2.5.3 Secure Operating System Architecture

The answer to the requirement of protection is an architecture that defines a fully and securely partitioned real-time operating system. The partitioning is carried out also in two dimensions: Spatial Partitioning and Temporal Partitioning.

In particular, the basic design of such an operating system complies with the design of an ordinary RTOS. The fundamental difference is located above the operating system's core layer within the application layer which in fact is a construction of several separate partitions of the ordinary application layer (cf. Fig. 2.11). Each partition is assigned to an integrity level only allowing the running of applications compliant to this level. Furthermore, it consists of a small Partition Operating System that provides operating system services according to the safety features required by the safety integrity level. Further-

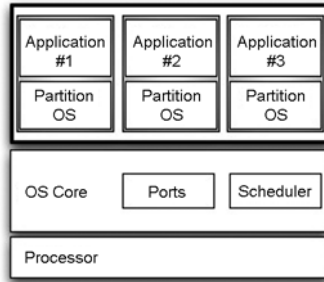


Figure 2.11. OS architecture for safety critical systems.

more, the Partition OS in fact runs the proper applications. The operating system core layer is responsible for the hardware-dependent functions, the device drivers, the scheduler, etc.

### 2.5.4 Providing Protection in Time Domain

The Scheduler implements temporal partitioning as it is responsible for assigning processor time to the partitions. Temporal partitioning requires an optimized two-level scheduler (cf. Fig. 2.12). The processor time for each partition is assigned statically. Within one scheduler period, also called major frame, each partition gets a guaranteed time window, a minor frame, to run its intrapartition processes. Within the minor frame, only the processes of the appendant partition can be executed. A partition is able to run more than one process. These processes have to be scheduled within the partition's processor

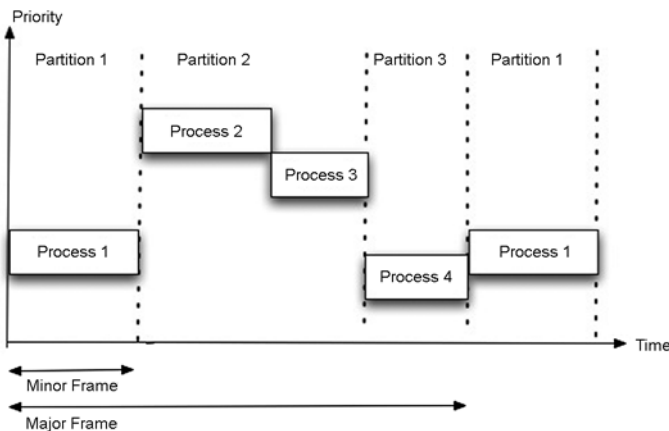


Figure 2.12. Two-level partition scheduler.

time frame. The partition remains the owner of the processor for the whole time frame, even if not all the processor time is needed for computation. Considering the given example above, a thread can only create a new thread within its own partition. Hence, the thread that creates the new thread has to share its time slice with the new thread without affecting the processor times of the other partitions.

### **2.5.5 Protection in Space Domain**

To avoid corruption of the data of a safety-critical application individual address spaces for processes are essential. Spatial partitioning is implemented by assigning one fragment of the entire memory to each partition. The memory space can only be accessed by the processes of that partition. Such a fragmentation of the memory requires the support of an integrated Memory Management Unit (MMU). When the scheduler switches between the minor frames, a new set of logical addresses is assigned to the memory manager. Hence, each partition can only access the logical address space that is mapped by the MMU which makes careless malicious corruption across the processes of different criticality levels impossible.

## **2.6 Multi-Core Architectures**

Multiprocessor architectures are an attempt to solve the lack of computational power in embedded systems by enabling computational concurrency. Using multiple lower-cost processors instead of cost-expensive high performance processors corresponds to the cost constraints of the embedded system market. However, multiprocessor architectures imply further challenges on the software and hence on operating systems that support these architectures [WJ04].

Multiprocessor architectures consist of multiple processing entities (PE) connected via an interconnection network. Each one of the processing entities may represent a microprocessor (central processing unit—CPU); it also may constitute any other hardware component such as a controller, decoder etc. There are several approaches for interconnecting the PEs but the typical ones are: shared bus, crossbar and micro network (network on chip). Depending on the type of interconnection a system shows up different performance (communication collisions), costs (e.g. chip area) and reliability (e.g. single-point-of-failure).

The design of an operating system that is applied in multiprocessor systems is strongly dependent on the underlying system architecture. The software design process is strongly coupled or even an inherent part of the hardware design. Basically, the operating system architecture for multiprocessor systems extends the architecture of uniprocessor operating systems: like in uniprocessor

processor operating systems it consists of the hardware abstraction layer and the core operating system. Furthermore, the operating system for multiprocessor application provides an inherent abstraction of the underlying system for the application. In fact, it abstracts design decisions of the multiprocessor architecture like:

- communication programming model: shared memory vs. distributed memory
- synchronous vs. asynchronous communication
- control strategy: centralized vs. decentralized
- redundancy mechanism
- hardware configuration
- topology: static vs. dynamic
- system architecture: homogeneity vs. heterogeneity

Beyond these design decisions that inherently incorporate into the RTOS implementation, resource management and scheduling, memory management, synchronization and interprocess communication (IPC) provide further challenges for a multi-core real time operating system.

### **2.6.1 Processor Management and Scheduling**

The processor management and the scheduling policy strongly depend on the design decisions of control strategy and the architectural design. The initial problem of the processor management is the assignment of processes to different processors. In the case of centralized control, the scheduling algorithm deals with NP-completeness. In homogeneous systems, each process can be assigned to any processor in the system whereas in heterogeneous architectures specific tasks can only be executed on specialized task-specific/application-specific system components. This architectural decision in turn affects also the complexity of a feasibility analysis. Multiprocessor systems enable real concurrency and hence task-level parallelism. One challenge of the scheduling policy is to enable processes belonging to one single job and having strong interaction, cooperation and communication in-between these processes to be executed simultaneously. Task Concurrency Management (TCM) addresses the dynamic and concurrent task scheduling problem of multiprocessor real-time operating systems. It introduces a two-phase scheduling method: design-time scheduling and run-time scheduling. An application is represented by a set of concurrent thread frames (TF) that consist of many thread nodes (TN), which are independent sections of code belonging to a single thread of control, the thread frame. At design-time, the scheduling is applied on each identified TN and results in a set of possible solutions that include different mappings, orderings and, as two-phase scheduling is a cost-oriented approach

(cost-performance, energy-oriented etc.), performance measures. From these possible solutions the design-time scheduler generates a Pareto-optimal set. However, to guarantee hard real-time requirements the schedules generated by the design-time exploration rely on worst-case conditions. Instead of dealing with the complex problem of computing schedules at run-time, the run-time scheduler operates on the TFs by determining one configuration of the Pareto curve established at design-time. Such a exploration at design-time significantly reduces computational cost at run-time. Details of the TNs mapping are invisible for the run-time scheduler which furthermore reduces its complexity.

### 2.6.2 Memory Management

Programming parallel processing applications raises two main questions: how do processes on different processors share data and how do these processes coordinate themselves? The answer to these questions in the first instance depends on the memory organisation of the system. We talk about distributed memory management if the processors possess private memories and about shared memory in case of a single address space. In the case of distributed memory management, data sharing and process cooperation is realized via message-passing. In contrast to that, shared memory management offers processors one single address space to share and exchange data. Shared memory systems require synchronisation mechanisms to prevent interferences between processes while operating on shared data.

### 2.6.3 Synchronisation

Processors in multiprocessor real-time systems require knowledge about the overall system time/clock. Therefore, synchronization of the global system clock is essential to ensure time-dependent performance. Due to dependability reasons, distributed clock synchronization mechanisms are preferred for multiprocessor RTOS as they do not provide a single-point-of-failure. There exist some approaches to ensure the synchronization of the global system time like: Time-Triggered Protocol (TTP), TT-Ethernet and FlexRay (in the automotive industry) [Par07]. TTP is a protocol for fault-tolerant communication between distributed real-time systems. The synchronization of the clock is achieved in a masterless manner based on identifying time differences of arriving messages. To ensure a dependable communication, the communications controllers define exact time slices for sending and receiving per system node. The clock synchronization mechanism defined in FlexRay is similar to that one in TTP. Basically, the synchronization is processed by sending micro ticks between the processors. The main difference is that FlexRay enables the synchronization of heterogeneous processor clocks by identifying local deviations of receiving

micro ticks. Similar to TTP, the communication policy in FlexRay is implemented through predefined time slots.

RTEMS<sup>1</sup> is a known example for a multiprocessor real-time operating system. Furthermore, the automotive industry has defined OSEK-OS<sup>2</sup>, a standard for operating systems designed to operate on the numerous controllers that are nowadays installed in cars.

## 2.7 Operating Systems for Wireless Sensor Networks

Given the recent advances in wireless sensor network (WSN) technology, it is possible to construct low-cost and low-power miniature sensor devices that can be spread across a geographical area in order to monitor their physical environment. Consisting of nodes equipped with a small processing unit, memory, a sensor, a battery and a wireless communication device, WSNs enable a myriad of applications ranging from human-embedded sensing to ocean data monitoring. Since each single node has only constrained processing and sensing capabilities, coordination among devices is necessary.

Due to their specific nature, sensor networks have different requirements compared to standard systems, such as self-configuration, energy-efficient operation, collaboration, in-network processing, as well as, a useful abstraction to the application developer. Given these requirements, a WSN OS must have a very small footprint and, at the same time, it must provide a limited number of common services for application developers, such as hardware management of sensors, radios, task coordination, power management, etc. (see [Sto05]). In the following section, we discuss some specific aspects relevant to OS for WSNs.

### 2.7.1 Aspects of Operating Systems for WSNs

We identify the following important aspects in WSNs:

**Hardware Management.** The OS should provide abstract services (e.g. for sensing and data delivery to neighbors). Given the lack of a memory management unit (MMU) in typical hardware, an OS library should implement this functionality (for more details see, e.g. [SRS<sup>+</sup>05]).

**Task Coordination.** There are two task coordination approaches:

- *Event-based Kernels:* Tasks are implemented as event handlers that run until completion. This enables concurrency without the need to elaborate mechanisms like per-thread stacks or mutual exclusion. The main advan-

---

<sup>1</sup><http://www.rtems.com>

<sup>2</sup><http://www.osek-vdx.org>

tage of this approach is its small memory footprint: because processes cannot block, just a global stack is necessary. However, a major problem occurring is the difficulty to implement applications with state-driven programming: the event-driven model is hard to manage by developers and not all problems are easily described as state machines. Further, interleaved concurrency is hard to realize in such systems.

- *Preemptive Thread Multitasking Kernels*: Preemption leads to the necessity of saving the current state of the registers to the stack. The necessity of one stack per thread leads to a relatively high memory footprint. Moreover, the context switch operation is rather time-consuming, i.e. for a task set composed mainly of IO-bound tasks or small tasks, the overhead caused by the context switch is relatively high. This problem can however be solved by assigning a static context to each process (as done e.g. in safety critical systems). In summary, given the resource constrained hardware of WSNs, the above points provide arguments against this OS paradigm. Nonetheless, preemptive multitasking supports the development of more complex, elaborate distributed applications and enables a straightforward porting of existing embedded applications.

**WSNOS Architecture.** Given the lack of MMUs in the typical WSN node hardware, the following OS architectures are predominantly employed (in contrast to e.g. monolithic kernel, microkernel or exokernel architectures in classical OS):

- *Library-based OS*: A set of functions implementing abstractions to facilitate the hardware management. Typically, it does not provide memory protection.
- *Component-based OS*: The OS consists of composable, self-contained components (also called “building blocks” or “modules”), which are, in contrast to library-based OS, interconnected via clear interfaces and interact with each other. They typically realize a well-defined function, such as the computation of a Cyclic Redundancy Check (CRC), and comprise code and state. Besides, the increased amount of modularity and configurability, this paradigm also suits the event-based programming approaches of WSNs. One example of a component-based OS is TinyOS, in which components are wired together explicitly using events for interaction (for details see [KW05]).

Often there are no clear borders between communication stack, OS services, and application. Cross-layer approaches are commonly used.

**Power Management.** Given the energy constraints of WSNs, different power management techniques have been developed (according to [DC05]):



- *Duty Cycling*: Reduces the average power utilization by cycling the power of a given subsystem.
- *Batching*: Amortizes the high cost of start-up by bundling several operations together and executing them in a burst.
- *Hierarchy Techniques*: Order the operations by their energy consumption and invoke the low-energy ones prior to the high-energy ones in a fashion similar to the short-circuit techniques used by several compilers for the evaluation of boolean expressions in various languages.
- *Redundancy reduction*: Using compression, aggregation or message suppression.

The low-power operation mode in WSNs can be addressed at various levels. In [DC05], the following levels have been recognized: sensing, communication, computation, storage, energy harvesting and reconfigurability support.

### 2.7.2 Examples of WSNOS

**TinyOS.** *TinyOS* [CHB<sup>+</sup>01] is a very efficient OS for WSNs that uses event-based task coordination in order to run on very resource-constrained nodes. The execution model is similar to a finite state machine. It consists of a set of components that are included in the applications when necessary. TinyOS addresses the main challenges of a sensor network: constrained resources, concurrent operations, robustness, and application requirement support.

Each TinyOS application consists of a scheduler and a graph of components. The components are described by their interface and internal implementation.

The concurrency model in TinyOS consists of a two-level scheduling hierarchy: events preempt tasks, but tasks do not preempt other tasks. Each task can issue commands or put other tasks to work. Events are initiated by hardware interrupts at the lowest levels. They travel from lower to higher levels and can signal events, call commands, or post tasks. Wherever a component cannot accomplish the work in a bounded amount of time, it should post a task to continue the work. This is because a non-blocking approach is implemented in TinyOS, where locks or synchronization variables do not exist. This means that components must terminate.

**Mantis Operating System (MOS).** The Mantis operating system (MOS) is a WSN OS designed to behave similarly to UNIX and provides a larger functionality than *TinyOS*. It is a lightweight and energy-efficient multithreaded OS for sensor nodes.

In contrast to TinyOS, the MANTIS kernel uses a priority-based thread scheduling with round-robin semantics within one priority level. To avoid race conditions within the kernel, binary and integer semaphores are supported.

The OS offers a multiprogramming model similar to that present in conventional OS, i.e., the OS complies with the traditional POSIX-based multithreading paradigm. All threads coexist in the same address space. The existence of multiple stacks (one per thread) makes MOS more resource-intensive than single-threaded OS (e.g. TinyOS).

The kernel of Mantis OS also provides device drivers and a network stack. The network stack is implemented using user-level threads and focuses on the efficient use of the limited memory.

**Contiki.** The *Contiki* [DGV04] operating system provides dynamic loading and unloading of programs and services during run-time. It also supports dynamic downloading of code enabling the software upgrade of already deployed nodes. All this functionality is offered at a moderate price: the system uses more memory than TinyOS but less than Mantis OS.

The main idea of *Contiki* is to combine the advantages of event-driven and preemptive multithreading in one system: the kernel of the system is event-driven, but applications desiring to use multithreading facilities can simply use an optional library module for that. A *Contiki* system is partitioned in core and loaded programs. This partition is determined at compilation time. The core comprises the kernel, program loader, run time libraries, and communication system.

## 2.8 Real-Time Requirements of Multimedia Application

The timing constraints for multimedia traffic originate from the requirement

- to maintain the same temporal relationship in the sequence of information on transmission from service provider to service requester
- from the necessity of preferably low offset delays between information departure and arrival
- the requisite to keep multiple types of media in sync

Consequently, each piece of information needs to be transmitted within a bound time frame and the traffic becomes real-time. Any failure to meet the timing constraints impairs the user-perceived Quality of Service (QoS) of networked multimedia applications. Different types of applications, however, have different QoS requirements. Common multimedia applications can be classified as *multimedia playback applications*, *streaming applications*, and *real-time interactive*.

- Multimedia playback applications transmit content that is pre-encoded and stored on a video server. A typical representative of this application is Video on Demand (VoD). As the video transmission is one-way and

does not involve conversational or low-latency bound elements, this type of application is tolerant to delays and delay variations.

- Streaming applications, as opposed to playback applications, require encoding video content on the fly as it is not available beforehand. This type of application does not involve conversational elements, but the latency of the transmission has a strong impact on the perceived user experience of the content. A typical candidate for a streaming application is Internet Protocol TV (IPTV) and presentable content covers live transmissions of sport events. Consequently, streaming applications have tighter requirements on delay bounds and delay variations.
- Real-time interactive applications exhibit the most challenging requirements with respect to delay and jitter. The interactive character of the applications requires conversational elements that often include speech and video. Video conferencing and interactive gaming are common representatives for this type of application. As the human perception is more sensitive to audio than it is to video it requires an undelayed synchronization between the two. Therefore, the delay bounds for this type of application are even more stringent than those for streaming applications.

QoS denotes a collective assemblage of components that (1) transform the qualitative set of user and application requirements into quantifiable performance metrics for resource allocation and (2) enforce them along the network path between a service requester and a service provider [Dit08]. Common performance metrics include the network bandwidth and acceptable bounds for packet loss, delay, and jitter. The key to QoS enforcement is to *differentiate* traffic into *isolated* transmission queues and *provide resources* on a per-flow (Intserv) or per-class (Diffserv) basis. This is accomplished by QoS traffic control and its approaches for call admission control (CAC), traffic classification, traffic shaping & policing, packet queueing, and packet scheduling. The cohesions of the individual approaches are depicted in Fig. 2.13.

The purpose of CAC is to protect traffic in a shared network by determining if an additional traffic flow's request for resources can be approved without causing interference to the resource allocation of admitted flows. It relies on a flow's traffic characterization that describes its performance metrics. A traffic classifier investigates packets for their priority level and forwards them to respective transmission queues that implement the traffic differentiation and isolation. Traffic policing and shaping ensure that flows conform to their traffic characterization, thereby defending the network against unexpected traffic bursts. The differentiated transmission queues are served by a scheduler according to a predefined scheduling policy that ensures the resource enforcement.

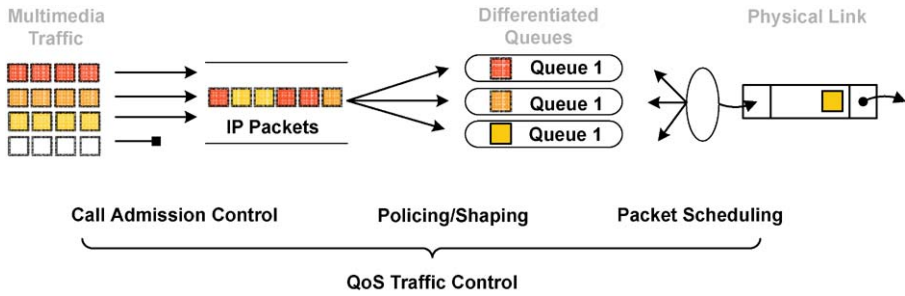


Figure 2.13. QoS Traffic Control Approaches.

The transmission of multimedia traffic across wireless networks imposes new and unique design challenges to the QoS traffic control and requires leveraging of the interaction among individual QoS approaches (1) toward lower layers of the communication model to optimize the resource provisioning of the scarce network resources and (2) in direction of the higher layers to perform content adaptation with different levels of granularity. The new challenges trace back to imperfect wireless transmission channels and the highly fluctuating traffic loads of multimedia applications. They are addressed by uni-directional cross-layer management approaches which can be partitioned into cross-layer optimization and cross-layer adaptation. Cross-layer optimization targets to improve the utilization or throughput of multimedia traffic in wireless networks by exploiting the time-varying channel characteristics. Cross-layer adaptation adjusts the content quality in respect to the traffic load toward the higher layers of the communication model. Popular approaches for multiple layer adaptation are Joint Source Channel Coding (JSCC) concepts [KYF<sup>+</sup>05].

## 2.9 Conclusions

Embedded applications in most cases are bound to real-time constraints and are usually executed on top of a Real-time Operating System (RTOS). Real-time tasks have to be annotated with basic timing information in order to enable the underlying RTOS to manage them properly. Such parameters include arrival time, worst case execution time (WCET) and (relative or absolute) deadline, just to mention the most important ones. Explicitly providing these information distinct real-time applications from ordinary ones where such information (usually characterized as non-functional properties) is available only in implicit manner. Having such characteristics in hand, specific scheduling algorithms can be designed. Most real-time applications show periodic behavior. When possible, static cyclic schedules are calculated off-line. If more flexibility is needed on-line techniques are applied. These algorithms are bound to priorities which can be assigned statically as in the case of Rate

Monotonic (RM) or Deadline Monotonic (DM) priority assignment, or dynamically as in the case of Earliest Deadline First (EDF). The latter one can be applied to a-periodic tasks as well. Task sets that consist of both periodic tasks and a-periodic ones, are more complicated to handle. An approach for a unified management of such situations is the introduction of so called servers. A server in this context is a periodic task that offers its processor utilization for executing a-periodic tasks.

Designing a proper RTOS architecture needs some delicate decisions. The basic services like process management, inter-process communication, interrupt handling, or process synchronization have to be provided in an efficient manner making use of a very restricted resource budget. Various techniques like library-based approaches, monolithic kernels, microkernels, or virtual machines/exokernels have been developed, each of them dedicated to specific demands. The classical approach is given by monolithic kernels. They allow efficient handling of service requests. Microkernels export as many services as possible into user space, thus reducing the risk of kernel corruption. Library-based approaches are more or less kernel-less. They can be adapted precisely to the needs of applications to be supported. Recently exokernels did gain interest. They support safety requirements in an elegant manner based on their virtualization technique.

Safety critical application can be supported by separation of applications either in the time or the space domain. Dedicated RTOS architectures preferably follow the concept of virtual machines/exokernels. By providing separated address spaces (space domain) or strictly separated time frames in scheduling (time domain) the mutual influence of tasks is substantially reduced. Multi-core architectures need special techniques for process management, memory management, and synchronization. Especially scheduling needs consideration as most of the classical RT scheduling methods are proven to be optimal only for mono-processor systems. An excellent fundamental architecture for distributed real-time systems is provided by time-triggered architectures, making use of time-triggered communication protocols.

The upcoming Wireless Sensor Networks (WSN) generate special demands for RTOS support leading to dedicated solutions. The nodes of a WSN are equipped with extremely restricted resources. Due to power constraints they have to be inactive for a large fraction of time. This implies special demands concerning communication and synchronization. As a consequence of these special requirements dedicated RTOS concepts have been developed. Strictly even-based approaches (e.g. UCB's TinyOS) may serve as an example. However, a tendency towards more standard multi-threading execution models can be observed. Another special area is given by multimedia applications. Very high data rates under (soft) RT constraints have to be supported. Based on the used encoding techniques (e.g. MPEG) dedicated solutions can be created.

In such solutions the frames within an MPEG Group of Pictures (GoP) can be scheduled in such a way that the number of frames to be dropped can be reduced.

The RTOS layer in an embedded system provides interesting glue between the underlying HW and the applications to be executed. Designing a fully predictable service provider in a highly efficient manner and at the same time making use of minimal resources is really challenging. This challenge is still open despite the fact that impressive solutions have been found by the RT community.

## References

- [BP84] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Commun. ACM*, 27(1):42–52, 1984.
- [But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)*. Springer-Verlag Telos, Santa Clara, 2004.
- [But05] Giorgio C. Buttazzo. Rate monotonic vs. EDF: judgment day. *Real-Time Syst.*, 29(1):5–26, 2005.
- [CHB<sup>+</sup>01] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A network-centric approach to embedded software for tiny devices. In *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*, pages 114–130. Springer, Berlin, 2001.
- [DC05] P.K. Dutta and D.E. Culler. System software techniques for low-power operation in wireless sensor networks. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design*, pages 925–932. IEEE Computer Society, Washington, 2005.
- [DGV04] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki—a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE Computer Society, Washington, 2004.
- [Dij68] E.W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.
- [Dit99] Carsten Ditzel. *Towards Operating System Synthesis*. Phd thesis, Department of Computer Science, Paderborn University, Paderborn, Germany, 1999.

- [Dit08] M. Ditze. *Coordinated Cross-Layer Management of QoS Capabilities for Transmitting Multimedia Traffic across Wireless IEEE 802.11 Networks*. To be published as University of Paderborn PhD Thesis, 2008.
- [Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic, Norwell, 1997.
- [KW05] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. Wiley, New York, 2005.
- [KYF<sup>+</sup>05] A. Katsaggelos, Y. Eisenberg, F. Zhai, R. Berry, and T. Pappas. Advances in efficient resource allocation for packet-based real-time video transmission. *Proc. IEEE*, 93(1):288–299, 2005.
- [OW02] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 55–64. ACM, New York, 2002.
- [Par07] Dominique Paret. *Multiplexed Networks for Embedded Systems*. Wiley, New York, 2007.
- [PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.
- [SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, San Mateo, 2005.
- [SRS<sup>+</sup>05] Brian Shucker, Jeff Rose, Anmol Sheth, James Carlson, Shah Bhatti, Hui Daia, Jing Deng, and Richard Han. Embedded operating systems for wireless microsensor nodes. In *Handbook of Sensor Network: Algorithms and Architectures*. Wiley, New York, 2005.
- [Sta01] William Stallings. *Operating Systems*. Prentice Hall, Upper Saddle River, 2001.
- [Sto05] Ivan Stojmenovic, editor. *Handbook of Sensor Networks: Algorithms and Architectures*. Wiley, New York, 2005.
- [Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, 2001.
- [THB06] A.S. Tanenbaum, J.N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.
- [WJ04] Wayne Wolf and Ahmed Jerraya. *Multiprocessor Systems-On-Chips*. Morgan Kaufmann, San Mateo, 2004.



<http://www.springer.com/978-1-4020-9435-4>

Hardware-dependent Software

Principles and Practice

Ecker, W.; Müller, W.; Dömer, R. (Eds.)

2009, XII, 299 p., Hardcover

ISBN: 978-1-4020-9435-4