Chapter 4
System Software

In order to cope with the complexity of applications of embedded systems, reuse of components is a key technique. As pointed out by Sangiovanni-Vincentelli [476], software and hardware components must be reused in the platform-based design methodology (see p. 296). These components comprise knowledge from earlier design efforts and constitute **intellectual property** (IP). Standard software components that can be reused include system software components such as embedded operating systems (OSs) and **middleware**. The last term denotes software that provides an intermediate layer between the OS and application software. This chapter starts with a description of general requirements for embedded operating systems. This includes real-time capabilities as well as adaptation techniques to provide just the required functionality. Mutually exclusive access to resources can result in priority inversion, which is a serious problem for real-time systems. Priority inversion can be circumvented with resource access protocols. We will present three such protocols: the priority inheritance, priority ceiling, and stack resource protocols. A separate section covers the ERIKA real-time system kernel. Furthermore, we will explain how Linux can be adapted to systems with tight resource constraints. Finally, we will provide pointers for additional reusable software components, like hardware abstraction layers (HALs), communication software, and real-time data bases. Our description of embedded operating systems and of middleware in this chapter is consistent with the overall design flow (see also Fig. 4.1).

© The Author(s) 2021

P. Marwedel, *Embedded System Design*, Embedded Systems,
https://doi.org/10.1007/978-3-030-60910-8_4

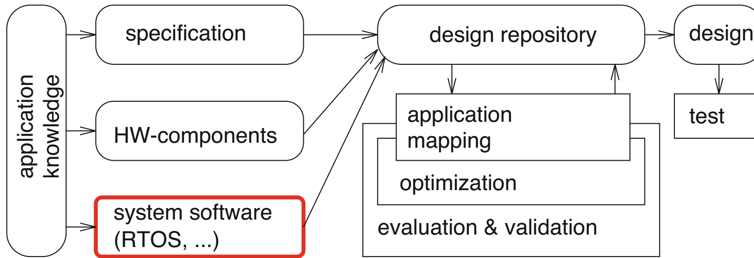


Fig. 4.1 Simplified design information flow

4.1 Embedded Operating Systems

4.1.1 General Requirements

Except for very simple systems, I/O, scheduling, and context switching require the support of an operating system suited for embedded applications. Switching from the execution of one code object such that some other code object is executed is called context switching. Context switching multiplexes processors such that each code object seems to have its own processor. For code objects, we distinguish between processes and threads. First of all, we define the term "process":

Definition 4.1 (Adopted from Tanenbaum [525]) A **process** is an executed program (or a part of a program) including memory content.

Courses on operating systems provide additional information about this term (e.g., in German [472]). In this chapter, we will be using this term in the sense of an entity within the operating system (and not in the sense of processes in SDL, VHDL, process networks, or semiconductor fabrication).

For systems with virtual addressing[1], we can distinguish between different address spaces. For such systems, we have to distinguish between executions of code objects within separate or within the same address spaces. If they are executed within separate address spaces, we will call them processes. If they are executed within the same address space, we will call them threads (or lightweight processes).

Definition 4.2 A **thread** is an executed program using the same address space as other programs.

For processes, there is some form of memory protection, since processes cannot corrupt other process memory areas. However, context switches have to change address translation information. Hence, they come with some overhead. For threads, this protection does not exist. In fact, threads sharing an address space will typically communicate via shared memory. Context switching for threads is typically faster

[1] See Appendix C.

than for processes. We do not need to distinguish between threads and processes if there is just one address space. More information about the just touched standard topics in system software can be found in textbooks on operating systems, such as the book by Tanenbaum [525]. Operating systems have to provide communication and synchronization methods for threads and processes.

The following are essential features of embedded operating systems:

- Due to the large variety of embedded systems, there is also a large variety of requirements for the functionality of embedded OSs. Due to efficiency requirements, it is not possible to work with OSs which provide the union of all functionalities. For most applications, the OS must be small. Hence, we need operating systems which can be **flexibly tailored** toward the application at hand. **Configurability** is therefore one of the main characteristics of embedded OSs. There are various techniques of implementing configurability, including:[2]
  - **Object orientation**, used for a derivation of proper subclasses: for example, we could have a general scheduler class. From this class we could derive schedulers having particular features. However, object-oriented approaches typically come with an additional overhead. For example, dynamic binding of methods does create run-time overhead. Ideas for reducing this overhead exist (see, e.g., https://github.com/lefticus/cppbestpractices/blob/master/08-Considering_Performance.md). Nevertheless, remaining overhead and potential timing unpredictability may be unacceptable for performance-critical system software.
  - **Aspect-oriented programming** [352]: with this approach, orthogonal aspects of software can be described independently and then can be added automatically to all relevant parts of the program code. For example, some code for profiling can be described in a single module. It can then be automatically added to or dropped from all relevant parts of the source code. The CIAO family of operating systems has been designed in this way [350].
  - **Conditional compilation**: in this case, we are using some macro preprocessor, and we are taking advantage of `#if` and `#ifdef` preprocessor commands.
  - **Advanced compile-time evaluation**: configurations could be performed by defining constant values of variables before compiling the OS. The compiler could then propagate the knowledge of these values as much as possible. Advanced compiler optimizations may also be useful in this context. For example, if a particular function parameter is always constant, this parameter can be dropped from the parameter list. Partial evaluation [275] provides a framework for such compiler optimizations. In a sophisticated form, dynamic data might be replaced by static data [26]. A survey of operating system specialization was published by McNamee et al. [387].
  - **Linker-based removal of unused functions**: at link-time, there may be more information about used and unused functions than during earlier phases. For

[2] This list is sorted by the position of the technique in the development process or tool chain.

example, the linker can figure out, which library functions are used. Unused library functions can be accordingly dropped and specializations can take place [91].

These techniques are frequently combined with a rule-based selection of files to be included in the operating system. Tailoring the OS can be made easy through a graphical user interface hiding the techniques employed for achieving this configurability. For example, VxWorks [590] from Wind River is configured via a graphical user interface.

Verification is a potential problem of systems with a large number of derived tailored OSs. Each and every derived OS must be tested thoroughly. Takada mentions this as a potential problem for eCos (an open-source RTOS; see http://ecos.sourceware.org and Massa [381]), comprising 100–200 configuration points [523]. For Linux, this problem is even larger [526]. Software product line engineering [456] can contribute toward solving this problem.

- There is a large variety of peripheral devices employed in embedded systems. Many embedded systems do not have a hard disk, a keyboard, a screen, or a mouse. There is effectively **no device that needs to be supported by all variants of the OS**, except maybe the system timer. Frequently, applications are designed to handle particular devices. In such cases, devices are not shared between applications, and hence there is no need to manage the devices by the OS. Due to the large variety of devices, it would also be difficult to provide all required device drivers together with the OS. Hence, it makes sense to decouple OS and drivers by using special processes instead of integrating their drivers into the kernel of the OS. Due to the limited speed of many embedded peripheral devices, there is also no need for an integration into the OS in order to meet performance requirements. This may lead to a different stack of software layers. For PCs, some drivers, such as disk drivers, network drivers, or audio drivers, are implicitly assumed to be present. They are implemented at a very low level of the stack. The application software and middleware are implemented on top of the application programming interface, which is standard for all applications. For an embedded OS, device drivers are implemented on top of the kernel. Applications and middleware may be implemented on top of appropriate drivers, not on top of a standardized API of the OS (see Fig. 4.2). Drivers may even be included in the application itself.
- **Protection mechanisms are sometimes not necessary**, since embedded systems are sometimes designed for a single purpose (they are not supposed to support

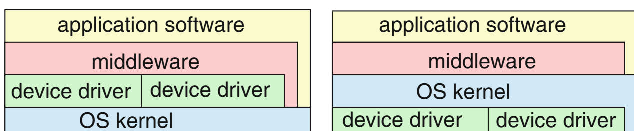


Fig. 4.2 Device drivers implemented on top of (**left**) or below (**right**) the OS kernel

the so-called multiprogramming). Untested programs have traditionally hardly ever been loaded. After the software has been tested, it could be assumed to be reliable. This also applies to input/output. In contrast to desktop applications, it is possibly not always necessary to implement I/O instructions as privileged instructions and processes can sometimes be allowed to do their own I/O. This matches nicely with the previous item and reduces the overhead of I/O operations.

Example 4.1 Let `switch` correspond to the (memory-mapped) I/O address of some switch which needs to be checked by some program. We can simply use a

```
load register,switch
```

instruction to query the switch. There is no need to go through an OS service call, which would create overhead for saving and restoring the context (registers, etc.). ∇

However, there is a trend toward more dynamic embedded systems. Also, safety and security requirements might make protection necessary. Special memory protection units (MPUs) have been proposed for this (see Fiorin [164] for an example). For systems with a mix of critical and non-critical applications (**mixed-criticality systems**), configurable memory protection [351] may be a goal.

- **Interrupts can be connected to any thread or process**. Using OS service calls, we can request the OS to start or stop them if certain interrupts happen. We could even store the start address of a thread or process in the interrupt vector address table, but this technique is very dangerous, since the OS would be unaware of the thread or process actually running. Also composability may suffer from this: if a specific thread is directly connected to some interrupt, then it may be difficult to add another thread which also needs to be started by some event. Application-specific device drivers (if used) might also establish links between interrupts and threads and processes. Techniques for establishing safe links have been studied by Hofer et al. [218].
- Many embedded systems are real-time (RT) systems, and, hence, the OS used in these systems **must be a real-time operating system** (RTOS).

Additional information about embedded operating systems can be found in a book chapter written by Bertolotti [51]. This chapter comprises information about the architecture of embedded operating systems, the POSIX standard, open-source real-time operating systems, and virtualization.

4.1.2 Real-Time Operating Systems

Definition 4.3 (A) *"real-time operating system is an operating system that supports the construction of real-time systems"* [523].

What is needed from an OS to be an RTOS? There are four key requirements:[3]

- **The timing behavior of the OS must be predictable.** For each service of the OS, an upper bound on the execution time must be guaranteed. In practice, there are various levels of predictability. For example, there may be sets of OS service calls for which an upper bound is known and for which there is not a significant variation of the execution time. Calls like "get me the time of the day" may fall into this class. For other calls, there may be a huge variation. Calls like "get me 4MB of free memory" may fall into this second class. In particular, the scheduling policy of any RTOS must be deterministic.

  There may also be times during which interrupts must be disabled to avoid interferences between components of the OS. Less importantly, they can also be disabled to avoid interferences between processes. The periods during which interrupts are disabled must be quite short in order to avoid unpredictable delays in the processing of critical events.

  For RTOSs implementing file systems still using hard disks, it may be necessary to implement contiguous files (files stored in contiguous disk areas) to avoid unpredictable disk head movements.
- **The OS must manage the scheduling of threads and processes**. Scheduling can be defined as mapping from sets of threads or processes to intervals of execution time (including the mapping to start times as a special case) and to processors (in case of multiprocessor systems). Also, the OS possibly has to be aware of deadlines so that the OS can apply appropriate scheduling techniques. There are, however, cases in which scheduling is done completely off-line and the OS only needs to provide services to start threads or processes at specific times or priority levels. Scheduling algorithms will be discussed in detail in Chap. 6.
- **Some systems require the OS to manage time**. This management is mandatory if internal processing is linked to an absolute time in the physical environment. Physical time is described by real numbers. In computers, discrete time standards are typically used instead. The precise requirements may vary:

  1. In some systems, synchronization with global time standards is necessary. In this case, **global clock synchronization** is performed. Two standards are available for this:

     - **Universal Time Coordinated (UTC)**: UTC is defined by astronomical standards. Due to variations regarding the movement of the Earth, this standard has to be adjusted from time to time. Several seconds have been added during the transition from 1 year to the next. The adjustments can be problematic, since incorrectly implemented software could get the impression that the next year starts twice during the same night.
     - **International atomic time** (in French: *temps atomic internationale* or TAI). This standard is free of any artificial artifacts.

[3]This section includes information from Hiroaki Takada's tutorial [523].

Some connection to the environment is used to obtain accurate time information. External synchronization is typically based on wireless communication standards such as the Global Positioning System (GPS) [413], mobile networks, or special atomic time services typically based on long wavelength stations [580], such as DCF77 in Germany.
2. If embedded systems are used in a network, it is frequently sufficient to synchronize time information within the network. Local clock synchronization can be used for this. In this case, connected embedded systems try to agree on a consistent view of the current time.
3. There may be cases in which provision for precise local delays is all that is needed.

For several applications, precise time services with a high resolution must be provided. They are required, for example, in order to distinguish between original and subsequent errors. For example, they can help to identify the power plant(s) that are responsible for blackouts (see [427]). The precision of time services depends on how they are supported by a particular execution platform. They are very imprecise (with precisions in the millisecond range) if they are implemented through processes at the application level and very precise (with precisions in the microsecond range) if they are supported by communication hardware. More information about time services and clock synchronization is contained in a book by Kopetz [303].

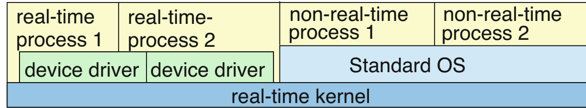
- **The OS must be fast**. An operating system meeting all the requirements mentioned so far would be useless if it were very slow. Therefore, the OS must obviously be fast.

Each RTOS includes a so-called real-time OS **kernel**. This kernel manages the resources which are found in every real-time system, including the processor, the memory, and the system timer. Major functions in the kernel include the process and thread management, interprocess synchronization and communication, time management, and memory management.

While some RTOSs are designed for general embedded applications, others focus on a specific area. For example, OSEK/VDX-compatible operating systems focus on automotive control. Operating systems for a selected area can provide a dedicated service for that particular area and can be more compact than operating systems for several application areas.

Similarly, while some RTOSs provide a standard API, others come with their own, proprietary API. For example, some RTOSs are compliant with the standardized POSIX RT-extension [201] for Unix, with the OSEK ISO 17356-3:2005 standard or with the ITRON specification developed in Japan (see http://www.ertl.jp/ITRON/). Many RT-kernel types of OSs have their own API. ITRON, mentioned in this context, is a mature RTOS which employs link-time configuration.

Fig. 4.3 Hybrid OSs



Available RTOSs can further be classified into the following categories [194]:

- **Fast proprietary kernels:** According to Gupta, *"for complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect"*. Examples include QNX, PDOS, VCOS, VTRX32, and VxWorks.
- **Real-time extensions to standard OSs:** In order to take advantage of comfortable mainstream operating systems, hybrid systems have been developed. For such systems, there is an RT-kernel running all RT-processes. The standard operating system is then executed as one of these processes (see Fig. 4.3).

  This approach has some advantages: for example, the system can be equipped with a standard OS API and can have graphical user interfaces (GUIs) and file systems. Enhancements to standard OSs become quickly available in the embedded world as well. Also, problems with the standard OS and its non-RT-processes do not negatively affect the RT-processes. The standard OS can even crash and this would not affect the RT-processes. On the down side, and this is already visible from Fig. 4.3, there may be problems with device drivers, since the standard OS will have its own device drivers. In order to avoid interference between the drivers for RT-processes and those for the other processes, it may be necessary to partition devices into those handled by RT-processes and those handled by the standard OS. Also, RT-processes cannot use the services of the standard OS. So all the nice features like file-system access and GUIs are normally not available to those processes, even though some attempts may be made to bridge the gap between the two types of processes without losing the RT capability. RT-Linux is an example of such hybrid OSs.

  According to Gupta [194], trying to use a version of a standard OS is *"not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation."* Indeed, dependencies between processes are not very frequent for most applications of standard operating systems and are therefore frequently ignored by such systems. This situation is different for embedded systems, since dependencies between processes are quite common and they should be taken into account. Unfortunately, this is not always done if extensions to standard operating systems are used. Furthermore, resource allocation and scheduling are rarely combined for standard operating systems. However, integrated resource allocation and scheduling algorithms are required in order to guarantee meeting timing constraints.

- There is a number of **research systems** which aim at avoiding the above limitations. These include Melody [569] and (according to Gupta [194]) MARS, Spring, MARUTI, Arts, Hartos, and DARK.

Takada [523] mentions low overhead memory protection, temporal protection of computing resources (targeting at preventing processes from computing for longer periods of time than initially planned), RTOSs for on-chip multiprocessors (especially for heterogeneous multiprocessors and multi-threaded processors), support for continuous media, and quality of service control as research issues.

Due to the potential growth in the Internet of Things (IoT) system market, vendors of standard OSs are offering variations of their products and obtain market shares from traditional vendors such as Wind River Systems [591]. Due to the increasing connectedness, Linux and its derivative Android® are becoming popular. Advantages and limitations of using Linux in embedded systems will be described in Sect. 4.4.

4.1.3 Virtual Machines

In certain environments, it may be useful to emulate several processors on a single real processor. This is possible with **virtual machines** executed on the bare hardware. On top of such a virtual machine, several operating systems can be executed. Obviously, this allows several operating systems to be run on a single processor. For embedded systems, this approach has to be used with care since the temporal behavior of such an approach may be problematic and timing predictability may be lost. Nevertheless, sometimes this approach may be useful. For example, we may need to integrate several legacy applications using different operating systems on a single hardware processor. A full coverage of virtual machines is beyond the scope of this book. Interested readers should refer to books by Smith et al. [502] and Craig [114]. PikeOS is an example of a virtualization concept dedicated toward embedded systems [520]. PikeOS allows the system's resources (e.g., memory, I/O devices, CPU-time) to be divided into separate subsets. PikeOS comes with a small micro-kernel. Several operating systems, application programming interfaces (APIs), and run-time environments (RTEs) can be implemented on top of this kernel (see Fig. 4.4).

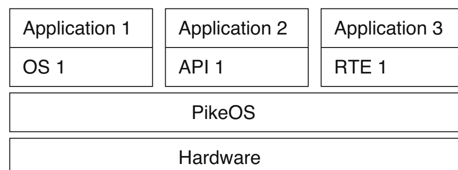


Fig. 4.4 PikeOS virtualization (©SYSGO)

4.2 Resource Access Protocols

In this section, we will be using the term **job**.

Definition 4.4 A particular execution of a (possibly repeatedly executed) task is called a **job**.

Compared to processes and threads used in operating systems, jobs can be seen as a more abstract view of required computations. During the design procedure, jobs will have to be mapped to entities handled by the operating system. A more precise definition will be provided in Definition 6.1.

4.2.1 Priority Inversion

There are cases in which jobs must be granted exclusive access to resources such as global shared variables or devices in order to avoid non-deterministic or otherwise unwanted program behavior. Such exclusive access is very important for embedded systems, e.g., for implementing shared memory-based communication or exclusive access to some special hardware device. Program sections during which such exclusive access is required are called **critical sections**. Critical sections should be short. Operating systems typically provide primitives for requesting and releasing exclusive access to resources, also called **mutex primitives**. Jobs not being granted exclusive access must wait until the resource is released. Accordingly, the release operation has to check for waiting processes and resume the job of highest priority.

In this book, we will call the request operation or lock operation P(S) and the release or unlock operation V(S), where S corresponds to the particular resource requested. P(S) and V(S) are so-called **semaphore** operations. Semaphores allow up to n (with n being a parameter) threads or processes to use a particular resource protected by S concurrently. S is a data structure maintaining a count on how many resources are still available. P(S) checks the count and blocks the caller if all resources are in use. Otherwise, the count is modified and the caller is allowed to continue. V(S) increments the number of available resources and makes sure that a blocked caller (if it exists) is unblocked. The names P(S) and V(S) are derived from the Dutch language. We will use these operations only in the form of binary semaphores with $n = 1$, i.e., we will allow only a single caller to use the resource.

For embedded systems, dependencies between processes are the rule, rather than an exception. Also, the effective job priority of real-time applications is more important than for non-real applications. Mutually exclusive access can lead to priority inversion, an effect which changes the effective priority of processes. Priority inversion exists on non-embedded systems as well. However, due to the reasons just listed, the priority inversion problem can be considered a more serious problem in embedded systems.

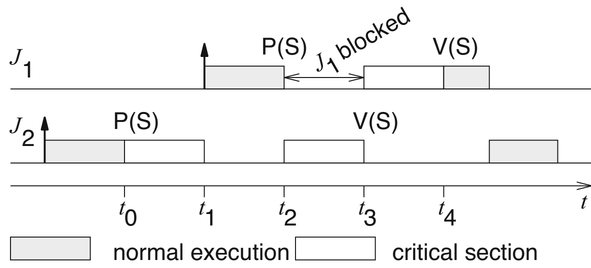


Fig. 4.5 Blocking of a job by a lower-priority job

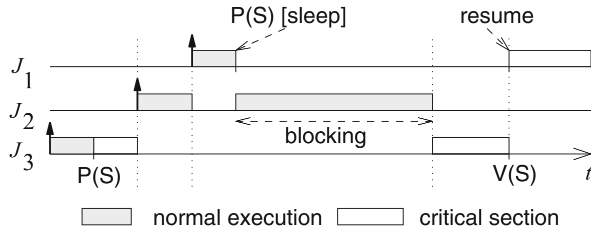


Fig. 4.6 Priority inversion with potentially large delay

A first case of the consequences resulting from the combination of "mutual exclusion" with "no preemption" can be seen in Fig. 4.5.

Bold upward pointing arrows indicate the times at which jobs are released or "ready". At time t_0, job J_2 enters a critical section after requesting exclusive access to some resource via an operation P. At time t_1, job J_1 becomes ready and preempts J_2. At time t_2, J_1 fails getting exclusive access to the resource in use by J_2 and becomes blocked. Job J_2 resumes and after some time releases the resource. The release operation checks for pending jobs of higher priority and preempts J_2. During the time J_1 has been blocked, a lower-priority job has effectively blocked a higher-priority job. The necessity of providing exclusive access to some resources is the main reason for this effect. Fortunately, in the particular case of Fig. 4.5, the duration of the blocking cannot exceed the length of the critical section of J_2. This situation is problematic but difficult to avoid.

In more general cases, the situation can be even worse. This can be seen, for example, from Fig. 4.6.

We assume that jobs J_1, J_2, and J_3 are given. J_1 has the highest priority, J_2 has a medium priority, and J_3 has the lowest priority. Furthermore, we assume that J_1 and J_3 require exclusive use of some resource via operation P(S). Now, let J_3 be in its critical section when it is preempted by J_2. When J_1 preempts J_2 and tries to use the same resource that J_3 is having exclusive access of, it blocks and lets J_2 continue. As long as J_2 is continuing, J_3 cannot release the resource. Hence, J_2 is effectively blocking J_1 even though the priority of J_1 is higher than that of J_2. In this example, the blocking of J_1 continues as long as J_2 executes. J_1 is blocked by a job of lower priority, which is not in its critical section. This effect is called **priority**

inversion.[4] In fact, priority inversion happens even though J_2 is unrelated to J_1 and J_3. The duration of the priority inversion situation is not bounded by the length of any critical section. This example and other examples can be simulated with the levi simulation software [497].

A prominent case of priority inversion happened in the Mars Pathfinder, where exclusive use of a shared memory area led to priority inversion on Mars [276].

4.2.2 Priority Inheritance

One way of dealing with priority inversion is to use the **priority inheritance protocol** (PIP). This protocol is a standard protocol available in many real-time operating systems. It works as follows:

- Jobs are scheduled according to their active priorities. Jobs with the same priorities are scheduled on a first-come, first-served basis.
- When a job J_1 executes P(S) and exclusive access is already granted to some other job J_2, then J_1 will become blocked. If the priority of J_2 is lower than that of J_1, J_2 inherits the priority of J_1. Hence, J_2 resumes execution. In general, every job inherits the highest priority of jobs blocked by it.
- When a job J_2 executes V(S), its priority is decreased to the highest priority of the jobs blocked by it. If no other job is blocked by J_2, its priority is reset to the original value. The highest priority job so far blocked on S is resumed.
- Priority inheritance is transitive: if J_x blocks J_y and J_y blocks J_z, then J_x inherits the priority of J_z.

This way, high-priority jobs being blocked by low-priority jobs propagate their priority to the low-priority jobs such that the low-priority jobs can release semaphores as soon as possible.

In the example of Fig. 4.6, J_3 would inherit the priority of J_1 when J_1 executes P(S). This would avoid the problem mentioned since J_2 could not preempt J_3 (see Fig. 4.7).

Figure 4.8 shows an example of nested critical sections [81]. Note that the priority of job J_3 is not reset to its original value at time t_0. Instead, its priority is decreased to the highest priority of the jobs blocked by it, in this case it remains at priority p_1 of J_1.

Transitiveness of priority inheritance is shown in Fig. 4.9 [81].

At time t_0, J_1 is blocked by J_2 which in turn is blocked by J_3. Therefore, J_3 inherits the priority p_1 of J_1.

Priority inheritance is also used by Ada: during a rendezvous, the priority of two threads is set to their maximum. Priority inheritance also solved the Mars Pathfinder

[4] Some authors do already consider the case of Fig. 4.5 as a case of priority inversion. This was also done in earlier versions of this book.

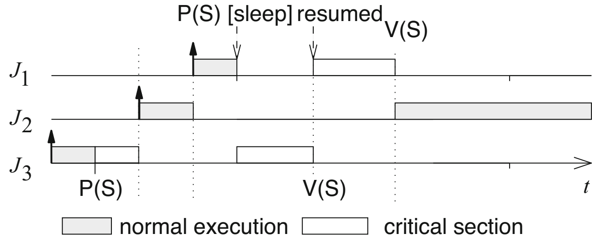


Fig. 4.7 Priority inheritance for the example of Fig. 4.6

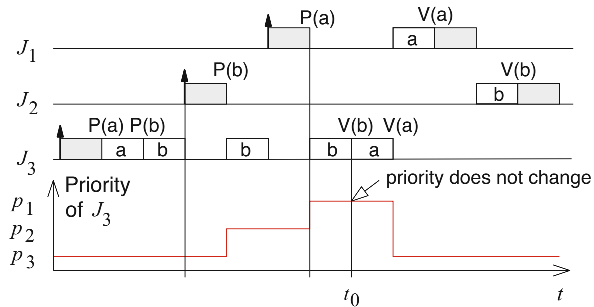


Fig. 4.8 Nested critical sections

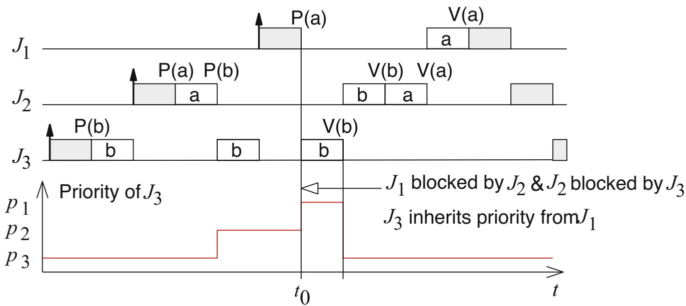


Fig. 4.9 Transitiveness of priority inheritance

problem: the VxWorks operating system used in the pathfinder implements a flag for the calls to mutex primitives. This flag allows priority inheritance to be set to "on." When the software was shipped, it was set to "off." The problem on Mars was corrected by using the debugging facilities of VxWorks to change the flag to "on," while the Pathfinder was already on Mars [276]. Priority inheritance can be simulated with the levi simulation software [497].

While priority inheritance solves some problems, it does not solve others. For example, there may be a large number of jobs having a high priority. There may also be deadlocks. The possible existence of deadlocks can be shown by means of an example [81]. Suppose that we have two jobs J_1 and J_2:

- For job J_1 we assume a code sequence of the form . . . ; P(a); P(b); V(b); V(a); . . . ;
- For job J_2 we assume a code sequence of the form . . . ; P(b); P(a); V(a); V(b); . . . ;.

Fig. 4.10 Priority inheritance deadlock

A possible execution sequence for these two jobs is shown in Fig. 4.10.

We assume that the priority of J_1 is higher than that of J_2. Hence, J_1 preempts J_2 at time t_1 and runs until it calls P(b), while b is held by J_2. Hence, J_2 resumes. However, it runs into a deadlock when it calls P(a). Such a deadlock would also exist if we were not using any resource access protocol.

4.2.3 Priority Ceiling Protocol

Deadlocks can be avoided with the **priority ceiling protocol** [485] (PCP). PCP requires jobs to be known at design time. With PCP, a job is not allowed to enter a critical section if there are already locked semaphores which could block it eventually. Hence, once a job enters a critical section, it cannot be blocked by lower-priority jobs until its completion. This is achieved by assigning a priority ceiling. Each semaphore S is assigned a priority ceiling C(S). It is the static priority of the highest-priority job that can lock S.

PCP works as follows:

- Let us assume that some job J is running and wants to lock semaphore S. Then, J can lock S only if the priority of J exceeds the priority ceiling C(S') of semaphore S' where S' is the semaphore with the highest-priority ceiling among all the semaphores which are currently locked by jobs other than J. If such a semaphore exists, then J is said to be blocked by S' and the job currently holding S'. When J gets blocked by S', the job currently holding S' inherits the priority of J.
- When some job J leaves a critical section guarded by S, it unlocks S and the highest-priority job, if any, which is blocked by S is awakened. The priority of J is set to the highest priority among all the jobs which are still blocked by some semaphore which J is still holding. If J is not blocking any other job, then the priority of J is set to its normal priority.

Figure 4.11 shows an example [59]. In this example, semaphores a, b, and c are used. The highest priority of a and b is p_1, and the highest priority of c is p_2.

At time t_2, J_2 wants to lock c, but c is already locked. Furthermore, the priority of J_2 does not exceed the ceiling of c. Nevertheless, the attempt to lock c results in an increase of the priority of J_3 to p_2.

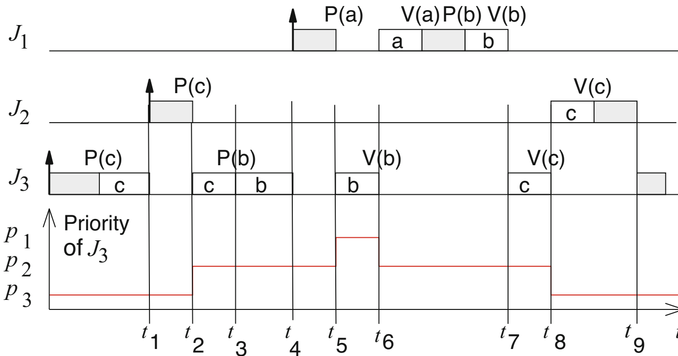


Fig. 4.11 Locking with the priority ceiling protocol

At time t_5, J_1 tries to lock a. a is not yet locked, but J_3 has locked b and the current priority of J_1 does not exceed the ceiling for b. So, J_1 gets blocked. This is the key property of PCP: this blocking avoids potential later deadlocks. J_3 inherits the priority of J_1, reflecting that J_1 is waiting for the semaphore b to be released by J_3.

At time t_6, J_3 unlocks b. J_1 is the highest-priority job so far blocked by b and now awakened. The priority of J_3 drops to p_2. J_1 locks and unlocks a and b and runs to completion. At time t_7, J_2 is still blocked by c, and for all jobs with priority p_2, J_3 is the only one that can be resumed. At time t_8, J_3 unlocks c and its priority drops to p_3. J_2 is no longer blocked, it preempts J_3 and locks c. J_3 is only resumed after J_2 has run to completion.

Let us consider a second example, to be used later for comparison with an extended PCP. Figure 4.12 shows this second example [59]. The highest priority of all semaphores is the priority of J_1. At time t_2, there is a request by J_3 for semaphore c, but the priority of J_3 is lower than the ceiling for the already locked semaphore a, and J_4 inherits the priority of J_3. At time t_3, there is a request for b, but the priority of J_2 is again lower than for the ceiling of the already locked semaphore a, and J_4 inherits the priority of J_2. At time t_5, there is a request for a, but the priority of J_1 is not exceeding the ceiling for a, and J_4 inherits the priority of J_1. When J_4 releases a, no semaphore is blocked and its priority drops to its normal priority. At this time, J_1 has the highest priority and executes until it terminates. Remaining executions are determined by the regular priorities.

It can be proven that PCP prevents deadlocks (see [81], Theorem 7.3). There are certain variants of PCP with different times at which the priority is changed. The Distributed Priority Ceiling Protocol (DPCP) [466] and the Multiprocessor Priority Ceiling Protocol (MPCP) [465] are extensions of PCP for multiprocessors.

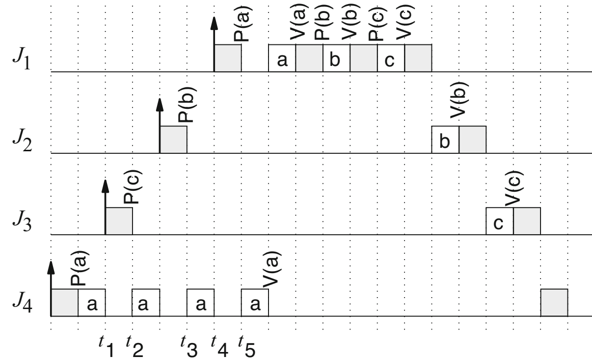


Fig. 4.12 Second PCP example

4.2.4 Stack Resource Policy

In contrast to PCP, the stack resource policy (SRP) supports dynamic priority scheduling, i.e., SRP can be used with dynamic priorities as computed by EDF scheduling (see Sect. 6.2.1 on p. 306). For SRP, we have to distinguish between jobs and tasks. Tasks may be describing repeating computations. Each computation is a job in the sense the term has been used so far. The notion of tasks captures features that apply to a set of jobs, e.g., the same code which needs to be executed periodically. Accordingly, for each task τ_i there is a corresponding set of jobs. See also Definition 6.1 on p. 297. SRP does not just consider each job of a task separately but defines properties which apply to tasks globally. Furthermore, SRP supports multi-unit resources, for example, memory buffers. The following values are defined:

- The **preemption level** l_i of a task τ_i provides information about which tasks can be preempted by jobs of τ_i. A task τ_i can preempt some other task τ_j only if $l_i > l_j$. We require that, if task τ_i arrives after τ_j and τ_i has a higher priority, then τ_i must have a higher preemption level than τ_j. For sporadic EDF scheduling (see p. 316), this means that the preemption levels are ordered inversely with respect to the relative deadlines. The larger the deadline, the easier it is to preempt the job. l_i is a static value.
- The **resource ceiling** of a resource is the highest preemption level of the tasks that could be blocked by issuing their maximum request for units of this resource. The resource ceiling is a dynamic value which depends on the number of currently available resource units.
- The **system ceiling** is the highest resource ceiling of all the resources which are currently blocked. This value is dynamic and changes with resource accesses.

SRP blocks the job at the time it attempts to preempt, instead of the time at which it tries to lock: a job can preempt another job if it has the highest priority and its preemption level is higher than the system ceiling. A job is not allowed to start until

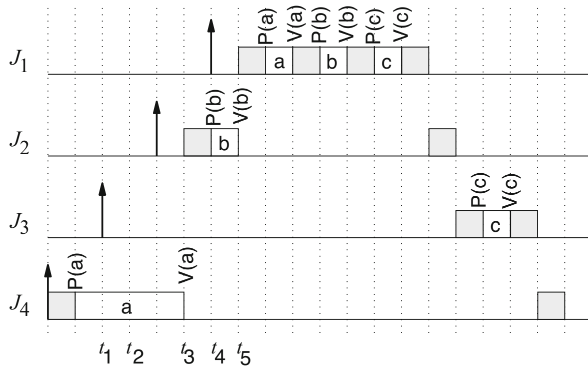


Fig. 4.13 SRP example

the resources currently available are sufficient to meet the maximum requirement of every job that could preempt it.

Figure 4.13 demonstrates the difference between PCP and SRP by means of the example shown in Fig. 4.12 [59]. For SRP, at time t_1 there is no preemption since the preemption level is not higher than the ceiling. The same happens at t_4. Overall, SRP has significantly less preemptions than PCP. This property has made SRP a popular protocol.

SRP is called *stack* resource policy, since jobs cannot be blocked by jobs with a lower l_i and can resume only when the job completes. Hence, jobs on the same level l_i can share stack space. With many jobs at the same level, a substantial amount of space can be saved.

SRP is also free of deadlocks (see Baker [34]). For more details about SRP, refer also to Buttazzo [81]. PIP, PCP, and SRP protocols have been designed for single processors. A first overview of resource access protocols for multiprocessors was published by Rajkumar et al. [466]. At the time of writing this book, there is not yet a standard resource access protocol for multi-cores (see Baruah et al. [41], Chapter 23).

4.3 ERIKA

Several embedded systems (such as automotive systems and home appliances) require the entire application to be hosted on small micro-controllers.[5] For that reason, the operating system services provided by the firmware on such systems must be limited to a minimal set of features allowing multi-threaded execution of periodic and aperiodic jobs, with support for shared resources to avoid the priority inversion phenomenon.

[5]This section was contributed by G. Buttazzo and P. Gai (Pisa).

Such requirements have been formalized in the 1990s by the OSEK/VDX Consortium [18], which defined the minimal services of a multi-threaded real-time operating system allowing implementations of 1–10 kilobytes of code footprint on 8 bit micro-controllers. The OSEK/VDX API has been recently extended by the AUTOSAR Consortium [28] which provided enhancements to support time protection, scheduling tables for time triggered systems, and memory protection to protect the execution of different applications hosted on the same micro-controller. This section briefly describes the main features and requirements of such systems, considering as a reference implementation the open-source ERIKA Enterprise real-time kernel [157].

The first feature that distinguishes an OSEK kernel from other operating systems is that all kernel objects are *statically* defined at compile time. In particular, most of these systems do not support dynamic memory allocation and dynamic creation of jobs. To help the user in configuring the system, the OSEK/VDX standard provides a configuration language, named OIL, to specify the objects that must be instantiated in the application. When the application is compiled, the OIL compiler generates the operating system data structures, allocating the exact amount of memory needed. This approach allows allocating only the data really needed by the application, to be put in flash memory (which is less expensive than RAM memory on most micro-controllers).

The second feature distinguishing an OSEK/VDX system is the support for *stack sharing*. The reason for providing stack sharing is that RAM memory is very expensive on small micro-controllers. The possibility of implementing a stack sharing system is related to how the code is written.

In traditional real-time systems, we consider the repetitive execution of code. A job corresponds to a single execution of the code. The code to be executed repeatedly is called a **task**. In particular, tasks may be periodically causing the execution of a job. The typical implementation of such a periodic task is structured according to the following scheme:

```
task(x) {
  int local;
  initialization();
  for (;;) {
    do_instance();
    end_instance();
  }
}
```

Such a scheme is characterized by a forever loop containing an instance (job) of the periodic task that terminates with a blocking primitive (end_instance()), which has the effect of blocking the task until the next activation. When following such a programming scheme (called *extended task* in OSEK/VDX), the task is always present in the stack, even during waiting times. In this case, the stack cannot be shared, and a separate stack space must be allocated for each task.

The OSEK/VDX standard also provides support for *basic tasks*, which are special tasks that are implemented in a way more similar to functions, according to the following scheme:

```
int local;
task x() {
  do_instance();
}
System_initialization() {
  initialization();
}
```

With respect to extended tasks, in basic tasks, the persistent state that must be maintained between different instances is not stored in the stack, but in global variables. Also, the initialization part is moved to system initialization, because tasks are not dynamically created, but they exist since the beginning. Finally, no synchronization primitive is needed to block the task until its next period, because the task is activated every time a new instance starts. Also, the task cannot call any blocking primitive; therefore it can either be preempted by higher-priority tasks or execute until completion. In this way, the task behaves like a function, which allocates a frame on the stack, runs, and then cleans the frame. For this reason, the task does not occupy stack space between two executions, allowing the stack to be shared among all tasks in the system. ERIKA Enterprise supports stack sharing, allowing all basic tasks in the system to share a single stack, so reducing the overall RAM memory used for this purpose.

Concerning task management, OSEK/VDX kernels provide support for fixed priority scheduling with Immediate Priority Ceiling to avoid the priority inversion problem. The usage of Immediate Priority Ceiling is supported through the specification of the resource usage of each task in the OIL configuration file. The OIL compiler computes the resource ceiling of each task based on the resource usage declared by each task in the OIL file.

OSEK/VDX systems also support non-preemptive scheduling and preemption thresholds to limit the overall stack usage. The main idea is that limiting the preemption between tasks reduces the number of tasks allocated on the system stack at the same time, further reducing the overall amount of required RAM. Note that reducing preemptions may degrade the schedulability of the tasks set; hence the degree of preemption must be traded off with the system schedulability and the overall RAM memory used in the system.

Another requirement for operating systems designed for small micro-controllers is *scalability*, which means supporting reduced versions of the API for smaller footprint implementations. In mass production systems, in fact, the footprint significantly impacts on the overall cost. In this context, scalability is provided through the concept of *conformance classes*, which define specific subsets of the operating system API. Conformance classes are also accompanied by an upgrade path between them, with the final objective of supporting partial implementation

of the standard with reduced footprint. The conformance classes supported by the OSEK/VDX standard (and by ERIKA Enterprise) are:

- BCC1: this is the smallest conformance class, supporting a minimum of eight tasks with different priority and one shared resource.
- BCC2: compared to BCC1, this conformance class adds the possibility to have more than one task at the same priority. Each task can have pending activations, that is, the operating system records the number of instances that have been activated but not yet executed.
- ECC1: compared to BCC1, this conformance class adds the possibility to have extended tasks that can wait for an event to appear.
- ECC2: this conformance class adds both multiple activations and extended tasks.

ERIKA Enterprise further extends these conformance classes by providing the following two conformance classes:

- EDF: this conformance class does not use a fixed priority scheduler but an Earliest Deadline First (EDF) Scheduler (see Sect. 6.2.1) optimized for the implementation on small micro-controllers.
- FRSH: this conformance class extends the EDF scheduler class by providing a resource reservation scheduler based on the IRIS scheduling algorithm [380].

Another interesting feature of OSEK/VDX systems is that the system provides an API for controlling interrupts. This is a major difference when compared to POSIX-like systems, where interrupts are an exclusive domain of the operating system and are not exported to the operating system API. The rationale for this is that on small micro-controllers users often want to directly control interrupt priorities; hence it is important to provide a standard way to deal with interrupt disabling/enabling. Moreover, the OSEK/VDX standard specifies two types of Interrupt Service Routines (ISR):

- Category 1: simpler and faster, does not implement a call to the scheduler at the end of the ISR
- Category 2: this ISR can call some primitives that change the scheduling behavior. The end of the ISR is a rescheduling point. ISR1 has always a higher priority of ISR2.

An important feature of OSEK/VDX kernels is the possibility to fine-tune the footprint by removing error-checking code from the production versions, as well as to define hooks that will be called by the system when specific events occur. These features allow for a fine-tuning of the application footprint that will be larger (and safer) when debugging and smaller in production when most bugs will be found and removed from the code.

To support a better debugging experience, the OSEK/VDX standard defines a textual language, named ORTI, which describes where the various objects of the operating system are allocated. The ORTI file is typically generated by the OIL compiler and is used by debuggers to print detailed information about operating

system objects defined in the system (e.g., the debugger could print the list of the tasks in an application with their current status).

All the features defined by the OSEK/VDX standard have been implemented in the open-source ERIKA Enterprise kernel [157], for a set of embedded microcontrollers, with a final footprint ranging between 1 and 5 kilobytes of object code. ERIKA Enterprise also implements additional features, like the EDF scheduler, providing an open and free-of-charge operating system that can be used to learn, test, and implement real applications for industrial and educational purposes.

4.4 Embedded Linux

Increasing requirements to the functionality of embedded systems, such as Internet connectivity (in particular for the Internet of Things) or sophisticated graphics displays, demand that a large amount of software is added to a typical embedded system's simple operating system. It has been shown that it is possible to add some of this functionality to small embedded real-time operating systems, e.g., by integrating a small Internet protocol (IP) network stack [142]. However, integrating a number of different additional software components is a complex task and may lead to functional as well as security deficiencies.

A different approach, enabled by the exponential growth of semiconductor densities according to Moore's law, is the adaptation of a well-tested code base with the required functionality to run in an embedded context. Here, Linux[6] has become the OS of choice for a large number of complex embedded applications following this approach, such as Internet routers, GPS satellite navigation systems, network-attached storage devices, smart television sets, and mobile phones. These applications benefit from easy portability—Linux has been ported to more than 30 processor architectures, including the popular embedded ARM, MIPS, and PowerPC architectures—as well as the system's open-source nature, which avoids the licensing costs arising for commercial embedded operating systems.

Adapting Linux to typical embedded environments poses a number of challenges due to its original design as a server and desktop OS. Below, we detail solutions available in Linux to tackle the most common problems that arise in its use in embedded systems.

4.4.1 Embedded Linux Structure and Size

Strictly speaking, the term "Linux" denotes only the kernel of a Linux-based operating system. To create a complete, working operating system, a number of additional

[6]This section on Embedded Linux was contributed by M. Engel (NTNU Trondheim).

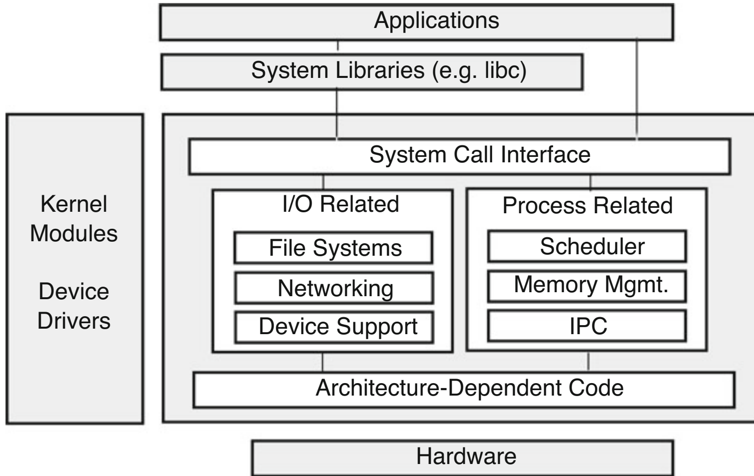


Fig. 4.14 Structure of typical Linux-based system

components are required that run on top of the Linux kernel. A configuration for a typical Linux system, including system-level user mode components, is shown in Fig. 4.14. On top of the Linux kernel reside a number of—commonly dynamically linked—libraries, which form the basis for system-level tools and applications. Device drivers in Linux are usually implemented as loadable kernel modules; however, restricted user mode access to hardware is also possible.

The open-source nature of Linux allows to tailor the kernel and other system components to the requirements of a given application and platform. This, in turn, results in a small system which enables the use of Linux in systems with restricted memory sizes.

One of the essential components of a Unix-like system is the C library, which provides basic functionality for file I/O, process synchronization and communication, string handling, arithmetic operations, and memory management. The libc variant commonly used in Linux-based systems is GNU libc (glibc). However, glibc was designed with server and desktop systems in mind and, thus, provides much more functionality than typically required in embedded applications. Linux-based Android® systems replace glibc with Bionic, a libc version derived from BSD Unix. Bionic is specifically designed to support systems running at lower clock speeds, e.g., by providing a tailored version of the Pthreads multithreading library to efficiently support Android's Dalvik Java VM. Bionic's size is estimated to be about half the size of a typical glibc version.[7]

Several significantly smaller implementations of libc exist, such as newlib, musl, uClibc, PDCLib, and dietlibc. Each of these is optimized for a specific use case; e.g.,

[7]The glibc-shared library size includes internationalization support.

libc version	musl	uClibc	dietlibc	glibc
Static library size	426 kB	500 kB	120 kB	2.0 MB
Shared library size	527 kB	560 kB	185 kB	7.9 MB
Minimal static C program size	1.8 kB	5 kB	0.2 kB	662 kB
Minimal static “Hello, World” size	13 kB	70 kB	6 kB	662 kB

Fig. 4.15 Size comparison of different Linux libc configurations

musl is optimized for static linking, uClibc was originally designed for MMU-less[8] Linux systems (see below), whereas newlib is a cross-platform libc also available for a number of other OS platforms. Sizes of the related shared library binary files range from 185 kB (dietlibc) to 560 kB (uClibc), whereas the glibc binary is 7.9 MB in size (all numbers taken from ×86 binaries) according to a comprehensive comparison of different libc implementation features and sizes, compiled by Eta Labs.[9] Figure 4.15 gives an overview of the sizes of various libc variants and programs built using the different libraries.

In addition to the C library, the functionality, size, and number of utility programs bundled with the OS can be adapted according to application requirements. These utilities are required in a Linux system to control system startup, operation, and monitoring; examples are tools to mount file systems, to configure network interfaces, or to copy files. As is the case for glibc, a typical Linux system includes a set of tools appropriate for a large number of use cases, most of which are not required on an embedded system.

An alternative to a traditional set of diverse tools is BusyBox, a software that provides a number of simplified essential Unix utilities in a single executable file. It was specifically created for embedded operating systems with very limited resources. BusyBox reduces the overhead introduced by the executable file format and allows code to be shared between multiple applications without requiring a library. A comparison of BusyBox with alternative approaches to provide a small user mode tool set can be found in [531].

4.4.2 Real-Time Properties

Achieving real-time guarantees in a system based on a general-purpose operating system kernel is one of the most complex challenges in adapting an OS to run in an embedded context. As shown above in Fig. 4.3, one common approach is to run the Linux kernel and all Linux user mode processes as a dedicated task of an underlying RTOS, only to be activated when no real-time task needs to run. In Linux, competing approaches exist that follow this design pattern. RTAI

[8] See Appendix C for an introduction to MMUs.

[9] Available online at http://www.etalabs.net/compare_libcs.html.

(real-time application interface) [138] is based on the Adeos hypervisor,[10] which is implemented as a Linux kernel extension. Adeos enables multiple prioritized domains (one of which is the Linux kernel itself) to exist simultaneously on the same hardware. On top of this, RTAI provides a service API, for example, to control interrupts and system timers. Xenomai [182] was co-developed with RTAI for several years but became an independent project in 2005. It is based on its own abstract "nucleus" RTOS core, which provides real-time scheduling, timer, memory allocation, and virtual file handling services. Both projects differ in their aims and implementations. However, they share the support for the Real-Time Driver Model (RTDM), a method to unify interfaces for developing device drivers and related applications in real-time Linux systems. The third approach using an underlying real-time kernel is RTLinux [608], developed as a project at the New Mexico Institute of Mining and Technology and then commercialized at the company FSMLabs, which was acquired by Wind River in 2007. The related product was discontinued in 2011. The use of RTLinux in products was controversial, since its initiators vigorously defended their intellectual property, for which they obtained a software patent [607]. The decision to patent the RTLinux methods was not well received by the Linux developer community, leading to spin-offs resulting in the abovementioned RTAI and Xenomai projects.

A more recent approach to add real-time capabilities to Linux, integrated into the kernel as of version 3.14 (2014), is SCHED_DEADLINE, a CPU scheduling policy based on the Earliest Deadline First (EDF) and Constant Bandwidth Server (CBS) [3] algorithms and supporting resource reservations. The SCHED_DEADLINE policy is designed to co-exist with other Linux scheduling policies. However, it takes precedence before all other policies to guarantee real-time properties.

Each task τ_i scheduled under SCHED_DEADLINE is associated with a runtime budget C_i and a period T_i, indicating to the kernel that C_i time units are required by that task every T_i time units, on any processor. For real-time applications, T_i corresponds to the minimum time elapsing between subsequent activations (releases) of the task, and C_i corresponds to the worst case execution time needed by each execution of the task. On addition of a new task to this scheduling policy, a schedulability test is performed and the task is only accepted if the test succeeds. During scheduling, a task is suspended when it tries to run for longer than the pre-allocated budget and deferred to its next execution period. This non work-conserving strategy[11] is required to guarantee temporal isolation between different tasks. Thus, on single-processor or partitioned multi-processor systems (with tasks pinned to a specific CPU), all accepted SCHED_DEADLINE tasks are guaranteed to be scheduled for an overall time equal to their budget in every time window as long as their period.

---

[10] See http://home.gna.org/adeos/.

[11] This means that the processor may be idle even when tasks could be executed. A definition of the term can be found in Chap. 6 on p. 309.

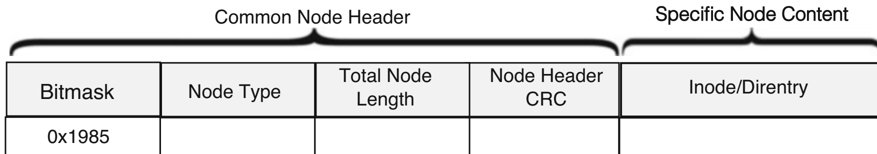


Fig. 4.16 Structure of the JFFS2 inode content

In the general case of tasks which are free to migrate on a multi-processor, as SCHED_DEADLINE implements global EDF (as described in detail in Sect. 6.3.3), the general tardiness bound for global EDF applies [128]. Benchmarks performed in [336] give an amount of missed deadlines of less than 0.2% when running SCHED_DEADLINE on a four-processor system with a utilization of 380% and 0.615% with a utilization of 390%. The numbers cited for a six-processor system are of similar magnitude. Of course, no deadline misses occur on single-processor systems or multi-core systems with processes pinned to a fixed processor core.

4.4.3 Flash Memory File Systems

Embedded systems pose different requirements to permanent storage than server or desktop environments. Often, there is a large amount of static (read-only) data, whereas the amount of varying data is in many cases quite limited.

Accordingly, file system storage can benefit from these special conditions. Since most of the read-only data in current embedded SoCs is implemented as flash ROM, optimization for this storage is an important aspect for the use of Linux in embedded systems. Accordingly, a number of different file systems specifically designed for using NAND-based flash storage have been developed.

One of the most stable flash-specific file systems available is the log-structured Journaling Flash File System version 2 (JFFS2) [596]. In JFFS2, changes to files and directories are "logged" to flash memory in so-called nodes. Two types of nodes exist, inodes (shown in Fig. 4.16), which consist of a header with file metadata followed by an optional payload of file data, and dirent nodes, which are directory entries each holding a name and an inode number. Nodes start out as valid when they are created and become obsolete when a newer version has been created in a different place in flash memory. JFFS2 supports transparent data compression by storing compressed data as inode payloads.

However, compared to other log-structured file systems such as Berkeley lfs [473], there is no circular log. Instead, JFFS2 uses blocks, a unit the same size as the erase segment of the flash medium. Blocks are filled with nodes in a bottom-up manner one at a time, as shown in Fig. 4.17.

Clean blocks contain only valid nodes, whereas dirty blocks contain at least one obsolete node. In order to reclaim memory, a background garbage collector collects

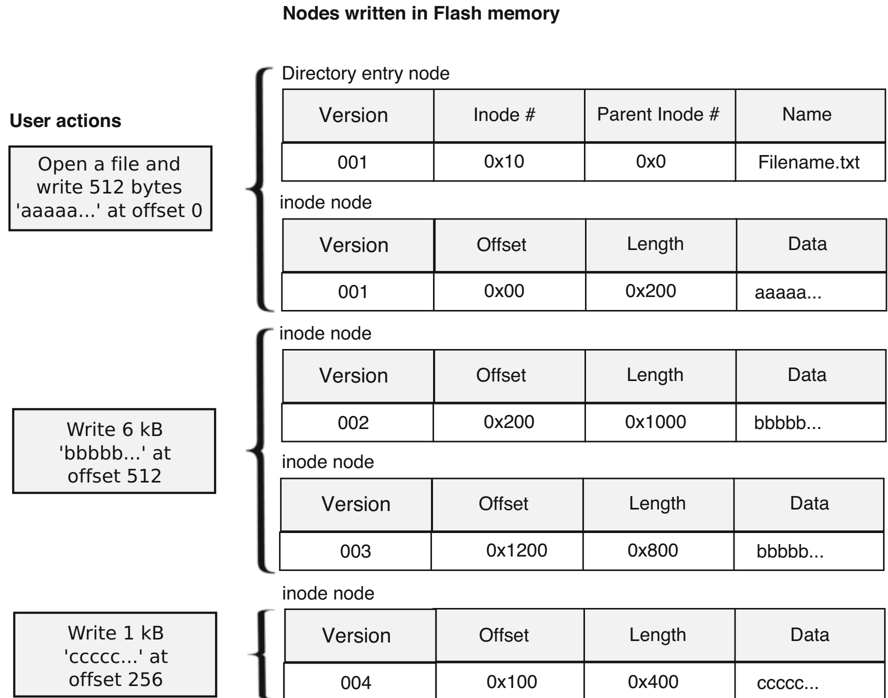


Fig. 4.17 Changes to flash when writing data to JFFS2

dirty blocks and frees them. Valid nodes from dirty blocks are copies into a new block, whereas obsolete blocks are skipped. After copying, the dirty block is marked as free. The garbage collector is also able to consume clean blocks in order to even out the flash memory wear-leveling and prevent localized erasure of blocks in a mostly static file system, as is common in many embedded systems.

4.4.4 Reducing RAM Usage

Traditionally, Unix-like operating systems treat main memory (RAM) as a cache for secondary storage on disk, i.e., swap space [385]. While this is a useful assumption for desktop and server systems with large disks and equally large memory requirements, it results in a waste of resources for embedded systems, since programs which exist in a system's non-volatile memory have to be loaded into volatile memory for execution. This commonly includes the rather large operating system kernel.

To eliminate this duplication of memory requirements, a number of execute-in-place (XiP) techniques have been developed which allow the direct execution of

program code from flash memory, which is the common approach in most smaller, microcontroller-based systems. However, XiP techniques face two challenges. On the one hand, the non-volatile memory storing the executable code needs to support accesses in byte or word granularity. On the other hand, executable programs are commonly stored in a data format such as ELF, which contains meta information (e.g., symbols for debugging) and needs to be linked at runtime before execution.

Support for XiP techniques is commonly implemented as a special file system, such as the Advanced XiP Filesystem (AXFS) [43], which provides compressed read-only functionality. The use of XiP is especially useful for the kernel itself, which would normally consume a large part of non-swappable memory. Running the kernel from flash memory would make more memory available for user-space code. XiP for user mode code itself is less useful, since the kernel only loads required text pages of an executable in virtual memory-enabled systems. Thus, RAM usage for program code is automatically minimized.

Providing the byte- or word-granularity accesses required for XiP is mostly a question of cost in current systems. The commonly used NAND flash technology, as used in flash disks, SD cards, and SSDs, is inexpensive but only allows block-level accesses, similar to hard disks. NOR flash is a flash technique supporting random accesses; thus it is suitable for implementing XiP techniques. However, NOR flash tends to be an order of magnitude more expensive than NAND flash and is commonly somewhat slower than system RAM. As a consequence, equipping a system with more RAM instead of a large NOR flash and not using XiP techniques is a sensible design choice for most systems.

4.4.5 uClinux: Linux for MMU-Less Systems

One final resource restriction is apparent in low-end microcontroller systems, such as ARM's Cortex-M series. The processor cores in these SoCs were developed for typical real-time OS scenarios, which often use a simple library OS approach, as described for ERIKA above. Thus, they lack crucial OS support hardware such as a paging memory management unit (see Appendix C). However, the large address space and relatively high clock speeds of these microcontrollers enable running a Linux-like operating system with some restrictions. Thus, uClinux was created as a derivative of the Linux kernel for MMU-less systems. Since kernel version 2.5.46, uClinux support is available in the mainstream kernel source tree for a number of architectures including ARM7TDMI, ARM Cortex-M3/4/7/R, MIPS, M68k/ColdFire, as well as FPGA-based softcores such as Altera Nios II, Xilinx MicroBlaze, and Lattice Mico32.

The lack of memory management hardware in uClinux-supported platforms comes with a number of disadvantages. An obvious drawback is the lack of memory protection, so any process is able to read and write other processes' memory. The lack of an MMU also has consequences for the traditional Unix process creation approach. Commonly, processes in Unix are created as a copy of an existing process

using the fork() system call [470]. Instead of creating a physical copy in memory, which would require copying potentially large amounts of data, only the page table entries of the process executing fork() are replicated and point to physical page frames of the parent process. When the newly created process memory starts to differ from its parent due to data writes, only the affected page frames are copied on demand using a copy-on-write strategy. The lack of hardware support for copy-on-write semantics and the overhead involved in actually copying pages result in the fork() system call being unavailable in uClinux.

Instead, uClinux provides the vfork() system call. This system call makes use of the fact that most Unix-style processes immediately call exec() after a fork to start a different executable file by overloading their memory image with text and data segments of that different binary:

```
pid_t childPID;
childPID = vfork();
if (childPID == 0) { // in child process
   execl("/bin/sh", "sh", 0);
}
printf("Parent program running again, child PID is %d", childPID);
```

The direct calling of exec() after vfork() implies that the complete address space of the newly created process will be replaced in any case and only a small part of the executable calling vfork() is actually used. In contrast to standard Unix behavior, vfork guarantees that the parent process is stopped after forking until the child process has called the exec() system call. Thus, the parent process is unable to interfere with the execution of the child process until the new program image has been loaded. However, some restrictions have to be observed to guarantee safe operation of vfork(). It is not permitted to modify the stack in the created child process, i.e., no function calls may be executed before exec. As a consequence, returning from vfork in case of an error, e.g., insufficient memory or inability to execute the new program, is impossible, since this would modify the stack. Instead, it is recommended to exit() from the child process in case of a problem.

To summarize, uClinux is a way to use some Linux functionality on low-end, microcontroller-style embedded systems. However, the on-chip memory even in high-end microcontrollers is restricted to several hundreds of kB. A minimal uClinux version, however, requires about 8 MB RAM, so the addition of an external RAM chip is essential. For systems offering a smaller memory footprint, more traditional RTOS systems are still the more feasible solution.

4.4.6 Evaluating the Use of Linux in Embedded Systems

In addition to technical criteria, the decision whether to base an embedded system on Linux also has to consider legal and business questions.

On the technical side, Linux includes support for a large number of CPU architectures, SoCs, and peripheral devices as well as communication protocols commonly used in embedded applications, such as Internet protocol TCP/IP, CAN, Bluetooth® or IEEE802.15.4/ZigBee®. It provides a POSIX-like API that enables easy porting of existing code, not only written in C or C++ but also in scripting languages such as Python or Lua and even more specialized languages like Erlang. Linux development tools are available free of charge and can easily be integrated into development toolflows utilizing IDEs such as Eclipse and continuous integration testing services such as Jenkins. While in general, the Linux code base is well tested, the quality of support varies with the targeted platform. When utilizing a less common hardware platform, it is recommended to thoroughly investigate the stability of CPU and driver support. One drawback of using Linux is the inherent complexity of the large code base, requiring a good insight into and experience with the system to debug problems. However, a number of semiconductor manufacturers and third-party companies offer commercial support for embedded Linux, including the provisioning of complete board support packages (BSPs) for a number of reference designs.

From a business perspective, the obvious benefit of using Linux is the availability of its source code free of cost. However, the GPL License version 2[12] governing the kernel source code also requires that the source code for modifications to the existing code base is provided along with the binary code. This might jeopardize trade secrets of hardware components or violate non-disclosure agreements with hardware intellectual property owners. For some hardware, such as GPU drivers, this is circumvented by the inclusion of binary code "blobs" which are loaded by an open-source device driver stub. However, this approach is being actively discouraged by the Linux kernel developers.

An increasingly serious problem is the security of embedded systems built on Linux, especially in the context of the Internet of Things. Many security problems affecting the Linux kernel also apply to embedded Linux. Inexpensive consumer devices, such as Internet-based cameras, routers, and mobile phones, rarely receive software updates but may be in active use for many years. This exposes them to security vulnerabilities which are already being actively exploited, e.g., for distributed denial-of-service attacks (DDOS) emanating from thousands of hijacked embedded Linux devices. As a consequence, the cost of continually updating devices in production as well as legacy devices in the field has to be considered in order to provide secure systems.

[12] See http://www.gnu.org/licenses/gpl-2.0.html.

4.5 Hardware Abstraction Layer

Hardware abstraction layers (HALs) provide a way for accessing hardware through a hardware-independent application programming interface (API). For example, we could come up with a hardware-independent technique for accessing timers, irrespective of the addresses to which timers are mapped. Hardware abstraction layers are used mostly between the hardware and operating system layers. They provide software intellectual property (IP), but they are neither part of operating systems nor can they be classified as middleware. A survey over work in this area is provided by Ecker, Müller, and Dömer [145].

4.6 Middleware

Communication libraries provide a means for adding communication functionality to languages lacking this feature. They add communication functionality on top of the basic functionality provided by operating systems. Due to being added on top of the OS, they can be independent of the OS (and obviously also of the underlying processor hardware). As a result, we will obtain communication-oriented cyber-physical systems. Such communication is needed for the Internet of Things (IoT). There is a trend toward supporting communication within some local system as well as communication over longer distances. The use of Internet protocols in general is becoming more popular. Frequently, such protocols enable **secure communication**, based on en- and decryption (see p. 196). The corresponding algorithms are a special case of middleware.

4.6.1 OSEK/VDX COM

OSEK/VDX® COM is a special communication standard for the OSEK automotive operating systems [441].[13] OSEK COM provides an "Interaction Layer" as an application programming interface (API) through which internal communication (communication within one ECU) and external communication (communication with other ECUs) can be performed. OSEK COM specifies just the functionality of the Interaction Layer. Conforming implementations must be developed separately.

The Interaction Layer communicates with other ECUs via a "Network Layer" and a "Data Link" layer. Some requirements for these layers are specified by OSEK COM, but these layers themselves are not part of OSEK COM. This way, communication can be implemented on top of different network protocols.

[13] OSEK is a trademark of Continental Automotive GmbH.

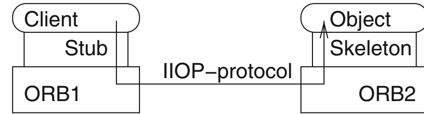


Fig. 4.18 Access to remote objects using CORBA

OSEK COM is an example of communication middleware dedicated toward embedded systems. In addition to middleware tailored for embedded systems, many communication standards developed for non-embedded applications can be adopted for embedded systems as well.

4.6.2 CORBA

CORBA® (Common Object Request Broker Architecture) [433] is one example of such adopted standards. CORBA facilitates the access to remote services. With CORBA, remote objects can be accessed through standardized interfaces. Clients are communicating with local stubs, imitating the access to the remote objects. These clients send information about the object to be accessed as well as parameters (if any) to the Object Request Broker (ORB; see Fig. 4.18). The ORB then determines the location of the object to be accessed and sends information via a standardized protocol, e.g., the IIOP protocol, to where the object is located. This information is then forwarded to the object via a skeleton, and the information requested from the object (if any) is returned using the ORB again.

Standard CORBA does not provide the predictability required for real-time applications. Therefore, a separate real-time CORBA (RT-CORBA) standard has been defined [428]. A very essential feature of RT-CORBA is to provide *end-to-end predictability of timeliness in a fixed priority system*. This involves *respecting thread priorities between client and server for resolving resource contention* and bounding the latencies of operation invocations. One particular problem of real-time systems is that thread priorities might not be respected when threads obtain mutually exclusive access to resources. The priority inversion problem (see p. 212) has to be addressed in RT-CORBA. RT-CORBA includes provisions for bounding the time during which such priority inversion can happen. RT-CORBA also includes facilities for thread priority management. This priority is independent of the priorities of the underlying operating system, even though it is compatible with the real-time extensions of the POSIX standard for operating systems [201]. The thread priority of clients can be propagated to the server side. Priority management is also available for primitives providing mutually exclusive access to resources. The priority inheritance protocol just described must be available in implementations of RT-CORBA. Pools of pre-existing threads avoid the overhead of thread creation and thread construction.

4.6.3 POSIX Threads (Pthreads)

The POSIX thread (Pthread) library is an application programming interface (API) to threads at the operating system level [37]. Pthreads are consistent with the IEEE POSIX 1003.1c operating system standard. A set of threads can be run in the same address space. Therefore, communication can be based on shared memory communication. This avoids the memory copy operations typically required for MPI (see Sect. 2.8.3 on p. 113). The library is therefore appropriate for programming multi-core processors sharing the same address space, and it includes a standard API with mechanisms for mutual exclusion. Pthreads use completely explicit synchronization [554]. The exact semantics depends on the memory consistency model used. Synchronization is hard to program correctly. The library can be employed as a back end for other programming models.

4.6.4 UPnP and DPWS

Universal Plug and Play (UPnP) is an extension of the plug-and-play concept of PCs toward devices connected within a network. Connecting network printers, storage space, and switches in homes and offices easily can be seen as the key target [438]. Due to security concerns, only data is exchanged. Code cannot be transferred.

Devices Profile for Web Services (DPWS) aims at being more general than UPnP. "*The Devices Profile for Web Services (DPWS) defines a minimal set of implementation constraints to enable secure Web Service messaging, discovery, description, and eventing on resource-constrained devices*" [597]. DPWS specifies services for discovering devices connected to a network, for exchanging information about available services, and for publishing and subscribing to events.

In addition to libraries designed for high-performance computing (HPC), several comprehensive network communication libraries can be used. These are typically designed for a loose coupling over Internet-based communication protocols.

MPI (see p. 113), OpenMP (see p. 114), OSEK/VDX COM, CORBA, Pthreads, UPnP, and DPWS are special cases of communication middleware (software to be used at a layer between the operating system and applications). Initially, they were essentially designed for communication between desktop computers. However, there are attempts to leverage the knowledge and techniques also for embedded systems. In particular, MPI (Message Passing Interface) is designed for message passing-based communication, and it is rather popular. It has recently been extended to also support-shared memory-based communication.

For mobile devices like smart phones, using standard middleware may be appropriate. For systems with hard time constraints (see Definition 1.8 on p. 10), their overhead, their real-time capabilities, and their services may be inappropriate.

4.7 Real-Time Databases

Databases provide a convenient and structured way of storing and accessing information. Accordingly, data bases provide an API for writing and reading information. A sequence of read and write operations is called a **transaction**. Transactions may have to be aborted for a variety of reasons: there could be hardware problems, deadlocks, problems with concurrency control, etc. A frequent requirement is that transactions do not affect the state of the database unless they have been executed to their very end. Hence, changes caused by transactions are normally not considered to be final until they have been **committed**. Most transactions are required to be **atomic**. This means that the end result (the new state of the database) generated by some transaction must be the same as if the transaction has been fully completed or not at all. Also, the database state resulting from a transaction must be **consistent**. Consistency requirements include, for example, that the values from read requests belonging to the same transaction are consistent (do not describe a state which never existed in the environment modeled by the database). Furthermore, to some other user of the database, no intermediate state resulting from a partial execution of a transaction must be visible (the transactions must be performed as if they were executed in **isolation**). Finally, the results of transactions should be persistent. This property is also called **durability** of the transactions. Together, the four properties printed in bold are known as ACID properties (see the book by Krishna and Shin [310], Chapter 5).

For some databases, there are soft real-time constraints. For example, time-constraints for airline reservation systems are soft. In contrast, there may also be hard constraints. For example, automatic recognition of pedestrians in automobile applications and target recognition in military applications must meet hard real-time constraints. The above requirements make it very difficult to guarantee hard real-time constraints. For example, transactions may be aborted various times before they are finally committed. For all databases relying on demand paging and on hard disks, the access times to disks are hardly predictable. Possible solutions include the main memory databases and predictable use of flash memory. Embedded databases are sometimes small enough to make this approach feasible. In other cases, it may be possible to relax the ACID requirements. For further information, see the book by Krishna and Shin as well as Lam and Kuo [319].

Table 4.1 Set of jobs requesting exclusive use of resources

Job	Priority	Arrival	Run-time	Printer		Comm line	
				$t_{P,P}$	$t_{V,P}$	$t_{P,C}$	$t_{V,C}$
J_1	1 (high)	3	4	1	4	–	–
J_2	2	10	3	–	–	1	2
J_3	3	5	6	–	–	4	6
J_4	4 (low)	0	7	2	5	–	–

4.8 Problems

We suggest solving the following problems either at home or during a flipped classroom session:

4.1 Which requirements must be met for an embedded operating system?

4.2 Which techniques can be used to customize an embedded operating system in the necessary way?

4.3 Which requirements must be met for a real-time operating system? How do they differ from the requirements of a standard OS? Which features of a standard OS like Windows or Linux could be missing in an RTOS?

4.4 How many seconds have been added at New Year's Eve to compensate for the differences between UTC and TAI since 1958? You may search in the Internet for an answer to this question.

4.5 Find processors for which memory protection units are available! How are memory protection units different from the more frequently used memory management units (MMUs)? You may search in the Internet for an answer to this question.

4.6 Describe classes of embedded systems for which protection should definitely be provided! Describe classes of systems, for which we would possibly not need protection!

4.7 Provide an example demonstrating priority inversion for a system comprising three jobs!

4.8 Download the levi learning module leviRTS from the levi web site [497]. Model a job set as described in Table 4.1.

$t_{P,P}$ and $t_{P,C}$ are the times relative to the start times, at which a job requests exclusive use of the printer or the communication line, respectively (called $\Delta t P$ in levi). $t_{V,P}$ and $t_{V,C}$ are the times relative to the start times at which these resources are released. Use priority-based, preemptive scheduling! Which problem occurs? How can it be solved?

4.9 Which resource access protocols prevent deadlocks caused by exclusive access to resources?

4.10 How is the use of the system stack optimized in ERIKA?

4.11 Which problems have to be solved if Linux is used as an operating system for an embedded system?

4.12 Which impact does the priority inversion problem have on the design of network middleware?

4.13 How could flash memory have an influence on the design of real-time databases?

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.