Architectures for ISO 26262 systems with multiple ASIL requirements

Yi Zheng, Product Manager, Safe and Secure Systems
QNX Software Systems

In-vehicle electronics

Roger Rivett, a functional safety specialist at Jaguar Land Rover, describes today's automobile nicely:

> "…rather than thinking of the vehicle as a mechanical machine, with some electrical components, it would be more accurate to think of the vehicle as a distributed computer system programmed for personal transport."[1]

The exponential increase in the number and complexity of in-vehicle electronics has transformed the automobile. At one time, the car was primarily an assembly of mechanical components; it has now become a system that integrates both mechanical and electronic components, with the electronic components representing a substantial portion of the added value and a disproportionate share of the headaches.

With a century of experience behind them, automakers have the building of the mechanical part of the car down to constant improvement and refinement of details. In-vehicle electronics, which include dozens of electronic control units (ECUs) and a head unit running complex infotainment software are a different matter. Not only are these systems evolving rapidly, but consumer demand for new applications and services is straining automakers' ability to deliver.

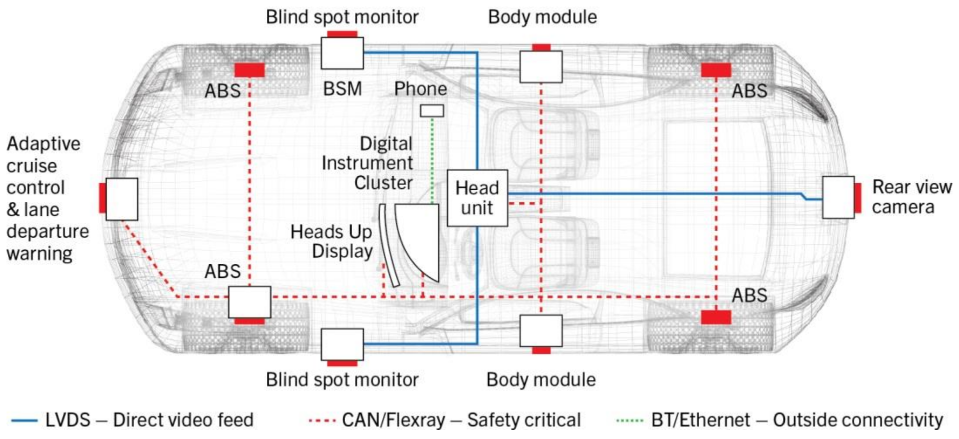

Figure 1. In-vehicle safety-related and non-safety-related systems distributed across different modules throughout the vehicle.

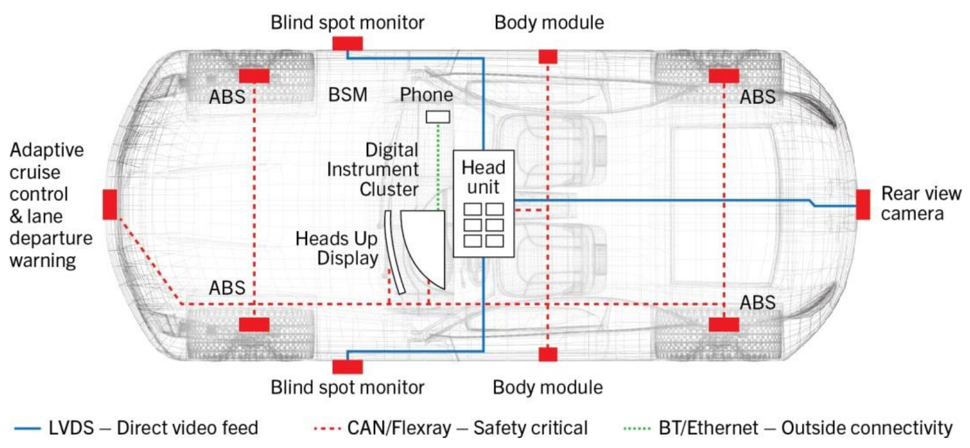

Figure 2. The same safety-related and non-safety-related systems shown in Figure 1, consolidated in the head unit.

Of course, automakers must provide all these new features without breaking the bank. The need to control costs, together with the availability of high-performance, low-cost processors, is driving consolidation of multiple in-vehicle systems onto one board. A design that eliminates one $50 module per vehicle translates into a substantial sum when multiplied by 5 million vehicles.

This consolidation creates its own challenges, however. In particular, many in-vehicle systems are safety-related, while others are consumer applications and impossible to prove as safe — yet all these disparate systems may need to run on the same CPU. Moreover, any in-vehicle system may now be connected, directly or indirectly, to the outside world. While this connectivity opens many new possibilities, such as over-the-air (OTA) firmware updates, it also creates new security and safety challenges.

The problem, then, is how to design and validate a system that incorporates components unlikely to require safety certification (for instance, a 3D display running consumer-grade applications) with components whose dependability and freedom from undesired interference must be rigorously engineered and proven (for instance, a blind spot detection module).

It is no accident that a main task set out by *ISO 26262 Road vehicles—Functional safety*[2] is the isolation of components.

ISO 26262 ASILs

Adapted from IEC 61508,[3] which specifies safety integrity levels according to probability of failure, ISO 26262 specifies four *automotive* safety integrity levels (ASILs). The lowest ASIL is A, the highest is D.

Unlike ISO 61508 SILs, which are for more general applications, ISO 26262 ASILs take into account the specifics of failures in an automobile. Each ASIL is based on a combination of three factors:

1. The probability that an event will occur.
2. The harm that will likely result from the event.
3. The probability that the driver will be able to control the vehicle following the event.

Components whose failure will result in an event that is unlikely to occur, unlikely to cause much harm, and unlikely to interfere with the driver's control of the vehicle may only require a safety level of ASIL A. Components whose failure will result in a common event that may cause great harm (such as serious injury or death) will likely require ASIL C or D, depending on the probability that the driver will lose control of the vehicle.

Interference

A violation of safety requirements occurs when ASIL components interfere with each other. The violation can occur when a component of any ASIL interferes with another component of a higher, equal, or even lower ASIL. Interference can take place when components are working together, or are supposed to work independently of each other. This is of particular concern when the components share a single CPU and memory subsystem.

For example, a communications module of ASIL B might interfere with an adaptive cruise control system of ASIL D. The communication module, in supplying data about ice on the road to the cruise control system, might rapidly broadcast an excessive number of messages (babbling), preventing the cruise control system from doing anything but receive messages. This same communications module might also interfere with a lower ASIL component such as the multimedia player by not releasing memory that the multimedia player needs to buffer music from the Internet.

Table 1 and Figure 3 summarize common forms of interference.

Interference	Description
Resource deprivation	By improperly using file descriptors, mutexes, flash memory, or other system resources, one process can deprive other processes of these resources. For example, by periodically using and not releasing a file descriptor, a process could eventually consume all the system's file descriptors and prevent another process from opening a file.
Time starvation	A process can prevent another process from completing its tasks by depriving it of computing time. For example, by performing a processor-intensive calculation or by entering a tight loop under a failure condition, a process could prevent a more critical process from running.
Illegal memory access	Occurs when a process reads or writes to the private memory of another process. A read access could constitute a security breach that leads to a safety problem later; a write access could immediately create a dangerous situation.
Data corruption	A process that shares corrupt data with another process may cause that process to behave in an unexpected and potentially unsafe manner.

Interference	Description
Babbling	A process may break its contract with a cooperating process and "babble" (send messages at a high rate or repeating messages) or send messages with incorrect data. Denial-of-service (DoS) attacks use this tactic to shut down a service, but such attacks are not the only source of babbling.
Deadlock	A deadlock occurs when cooperating processes wait for each other to complete. Since no process can advance until the other finishes, the system makes no forward progress. The circumstances that give rise to deadlocks are generally subtle and, because of their temporal nature, can seldom be detected or reproduced by testing.

Table 1. Common ways software components can interfere with the correct behavior of other software components.

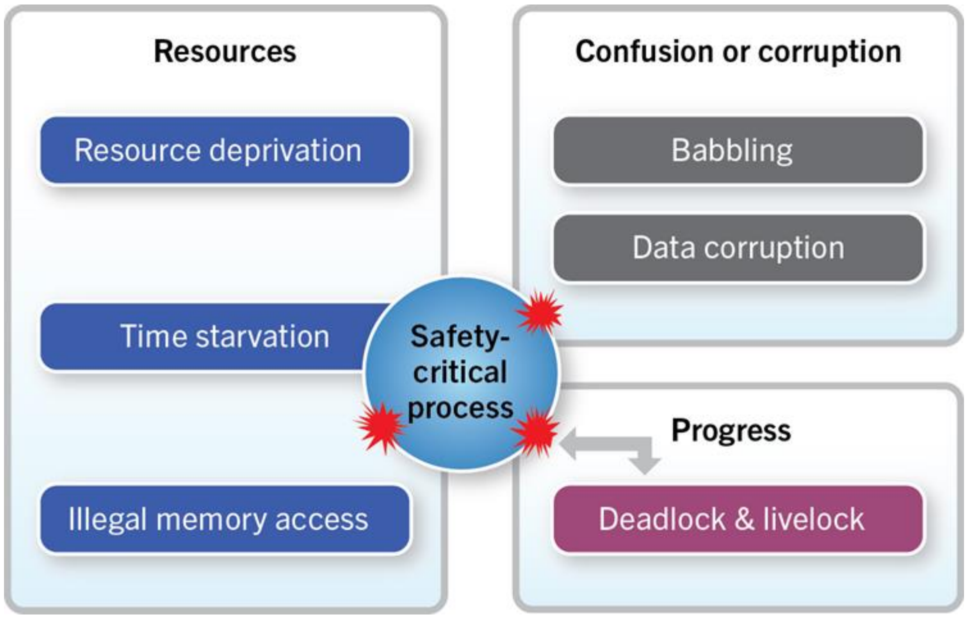

Figure 3. Ways in which other processes can interfere with the correct behavior of a safety-related process.

Building a resilient system

All design techniques have limitations and drawbacks. Fortunately, design only represents one line of defense. Techniques such as formal design and static analysis should also be used at appropriate stages in the project. And, as specified in ISO 26262, isolation of components from interference by components of different ASILs offers another technique for building resilient systems that can meet safety requirements.

Faults, errors, and failures

Paradoxically, a fundamental principle of safe system design is the recognition and acceptance that the system will contain faults. As Tom Anderson, a professor at Newcastle University's Centre for Software Reliability, wrote in *Safety Systems* journal:

> The inherent complexity of present-day software systems (including single-threaded), compounded by the vast range of possible input sequences with which such systems interact, leads to a pace of behavioral possibilities of enormous magnitude, a space where the notion of determinism becomes a matter of philosophy or even sophistry.[4]

Modern software systems have become so complex that is impossible to empirically prove them fault-free; that is, to test all possible paths and states. Automotive systems are no exception. Quoted in the IEEE's *Spectrum* magazine in 2009, Manfred Broy, a professor of informatics at the Technology University, Munich, noted that "a premium-class automobile '…probably contains close to 100 million lines of software code'." The authors of ISO 26262 make the point rather less dramatically, but no less accurately:

> With the trend of increasing technological complexity, software content and mechatronic implementation, there are increasing risks from systematic failures and random hardware failures. ISO 26262 includes guidance to avoid these risks by providing appropriate requirements and processes.[5]

Figure 4 below presents an adaptation of James Reason's model of how faults become errors, which lead to failures.[6] In short, something is bound to go wrong. Anyone building a safe system must keep this assumption in mind.

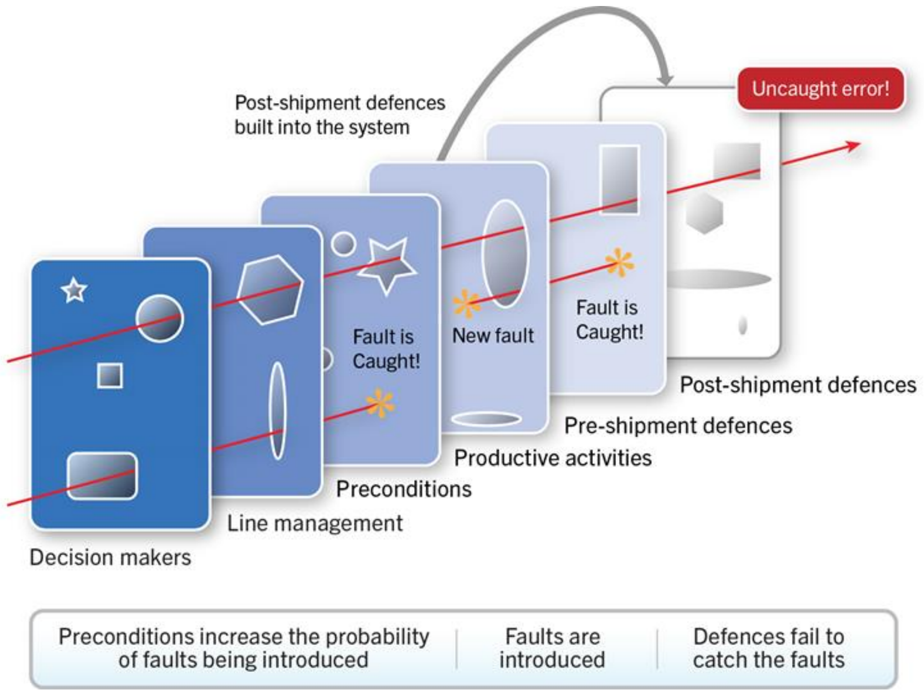

Figure 4. James Reason's model (adapted) of how faults become failures.

ISO 26262 acknowledges this problem through the importance it places on isolation. If software components—and especially components of different

ASILs—can be isolated from each other, then the failure of one component will be contained. The failure won't compromise other components or the entire system. Indeed, many errors, such as writing data into the memory of other processes, may not cause the component that contains the offending code to fail, but may interfere with another component and bring it or the entire system down.

Thus, a resilient system not only uses components that are sufficiently dependable to meet their respective safety requirements (their ASILs for automotive systems), but also isolates safety-related components from the effects of failures in other components. Current strategies for ensuring isolation involve virtualization and microkernel OS architecture.

Virtualization

Developers can choose from two main virtualization techniques: in a Type 1 hypervisor, the different guest OSs run on the virtualization layer, and in a Type 2 hypervisor, a guest OS runs nested inside another OS.

A hypervisor can help provide the component isolation required of an ISO 26262 system. Two OSs could run on the virtualization layer, each in a separate environment. One OS would run the safety-related components and the other would run everything else, such as multimedia applications and 3D navigation. Each OS would run as if it were the only OS on the board, using the resources allocated to it by the virtualization layer.

Things to consider when evaluating virtualization

Virtualization is attractive and seemingly simple, but there are many factors, both technical and financial, to consider before adopting a virtualization solution.

Visibility

Much of the functionality of the virtualization layer depends on the hardware. The hardware providing virtualization support is as complex as a memory management unit (MMU). But unlike MMU technology, which has now had years to prove itself in use, on-chip virtualization support is still relatively new. If a bug on a chip compromises dependability or software component isolation, either the chip has to be replaced, or work-arounds must be found and implemented in the hypervisor and possibly in the safety-related guest OS—all expensive undertakings.

Performance

Virtualization adds another layer of software to the system. New hardware technologies have gone a long way to minimize the latency introduced by the virtualization layer, but the virtualization layer itself may still affect performance of critical components. This can be especially problematic for hardware peripherals that require high bandwidth, such as a graphics processing unit (GPU). In a head unit that, for example, combines a low ASIL level infotainment system with a high ASIL level pedestrian-detection warning system, both systems could need to share high-bandwidth GPU resources. The virtualization layer would need to distribute the GPU resources intelligently so as to guarantee the smooth function of the pedestrian warning system.

Complexity

Hypervisor designs typically involve different OSs: one for the safety-related components and one for everything else. This, of course, is a more complex proposal than building a system with a single OS.

Granularity

Virtualization isolates the two OSs from each other, but it doesn't isolate components running on each guest OS. The OS running the safety-related components doesn't have to protect these components from non-safety-related components, but it must still be able to isolate safety-related components from each other.

For example, a system that includes an infotainment system, digital instrument cluster, adaptive cruise control system, and lane departure warning system might run the infotainment system on guest OS A and the other components on guest OS B. This approach would isolate the safety-related components from the infotainment system, but does nothing to protect, say, the lane departure warning from interference from the digital instrument cluster or adaptive cruise control. This protection would have to be handled by additional isolation and separation mechanisms within OS B, as specified in ISO 26262, Part 6, 7.4.11:

> If software partitioning… is used to implement freedom from interference between software components it shall be ensured that… the shared resources are used in such a way that freedom from interference of software partitions is ensured[7]

Figure 5 illustrates how, even with virtualization, OS B must provide partitioning to protect the safety-related components from each other.

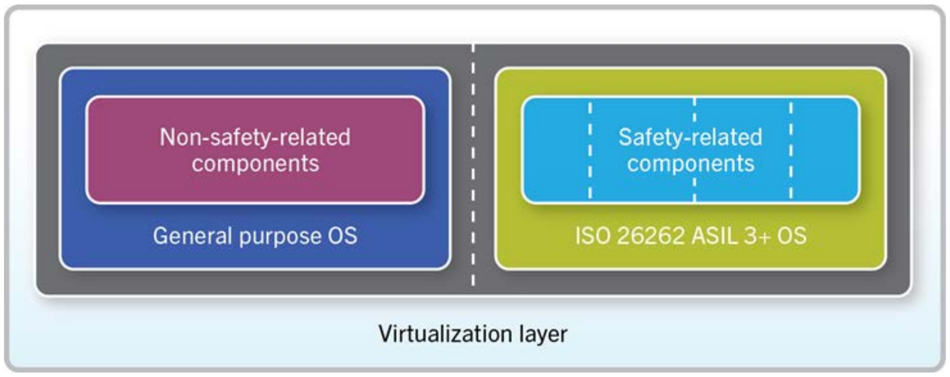

Figure 5. With virtualization, the OS running the safety-related components remains responsible for isolating these components from each other.

Long-term cost

The runtime licensing of a hypervisor commands its own share of the Bill of Materials (BOM). Thus, while a hypervisor-based design can reduce hardware costs, it does not necessarily reduce the overall BOM for the software.

More significantly, responsibility for an in-vehicle safety-related software system does not end when the vehicle rolls off the assembly line, but continues throughout the life of the vehicle. When something goes wrong, the automaker must remedy the situation. The automaker or its suppliers must be prepared, therefore, to maintain and assume the cost of the dual development infrastructures inherent in a two-OS solution throughout the lifespan of the vehicle.

Isolation and dependability

Whether or not virtualization is used to isolate safety-related components from non-safety-related components, an ISO26262 system must be designed so that:

- safety-related components meet their dependability requirements
- safety-related components are protected from interference from other components, both non-safety-related and safety-related

OS architectures

OS architecture is crucial in an ISO 26262 system, both because it is fundamental to overall system dependability and because it determines how easy it is to isolate and protect components with different or equivalent ASIL requirements. Table 2 below lists the most common OS architectures used in embedded systems and summarizes how these architectures affect component isolation.

OS Type	Design	Advantages	Disadvantages
Real-time executive	All components run together in a single memory address space	Efficient	A pointer error in one component can corrupt memory used by the kernel or by another component, causing system-wide failure.
Monolithic OS	Applications run as memory-protected processes. Kernel components share the same address space as file systems, protocol stacks, and drivers.	The kernel is protected from errant user code.	A fault in a device driver or other service that shares the same address space as the kernel can bring down the entire system.
Microkernel OS	Applications, device drivers, file systems, and networking stacks reside in separate address spaces, isolated from the kernel and each other.	Faults won’t percolate across the system. The system can restart a failed component. Components with different ASILs can be combined in the same system.	Small increase in overhead for inter-component communication.

Table 2. OS architectures and how they address component isolation.

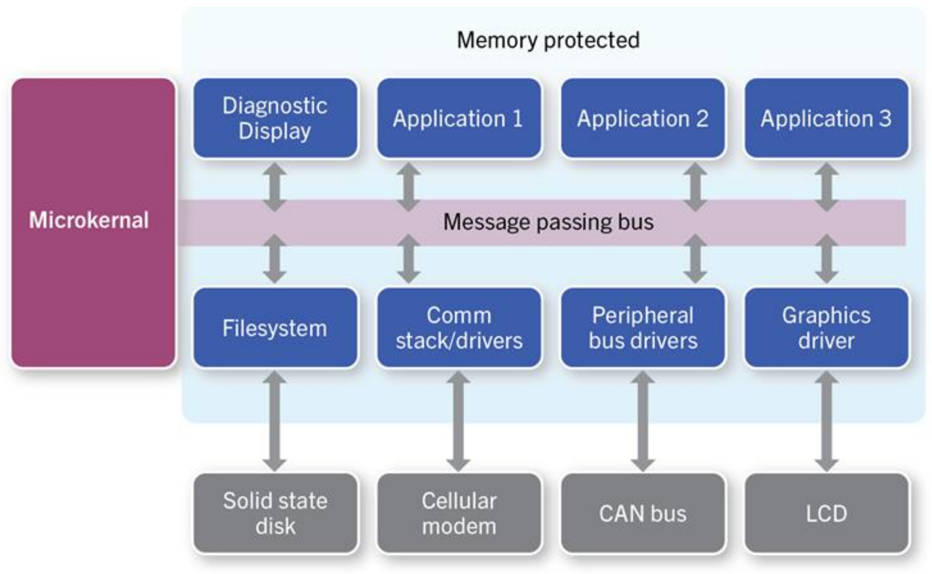

Figure 6. A microkernel OS isolates components from each other; a fault in one component can't percolate across the system.

An in-vehicle system will likely incorporate a multimedia component that uses high-end 3D graphics to display non-critical information on the head unit screen. This component may only require an ASIL of B or even A, while the safety-critical components (managing braking, adaptive cruise control, assisted parking, etc.) will require ASIL C certification or better. We suggest that a single microkernel OS can provide both sufficient dependability and sufficient protection from interference for an ISO 26262 system.

Protection from interference

In general, in a system with safety-related components, it is best to isolate as many components as possible, using a variety of complementary techniques. These techniques are applicable to different stages of the project, from design to validation of the completed components and system. The following OS features can help address the types of interference described in the "Interference" section, above.

Preventing resource deprivation

By using resource limit (`rlimit`) parameters, system designers can set upper limits on the size and quantity of resources allocated to a process or application (address space, memory, number of processes or threads, number of file descriptors, etc.). Thus, no process or application can monopolize resources and starve other processes.

To provide another line of defense, the system can include an anomaly detection program. This program would learn what constitutes normal behavior for a particular system, then monitor resource allocations and take corrective action when it detects that a process is making abnormal use of resources.

Bound multiprocessing (BMP) can also help protect resources needed by safety-related components. BMP is an advanced form of processor affinity—or symmetrical multiprocessing (SMP)—that lets designers assign threads or entire hierarchies of threads to specific cores. In an ISO 26262 system running on dual-core processor, Core A could be dedicated to threads for safety-related components, excluding all other threads, while Core B could run the threads for all the non-safety-related components. Thus, a multimedia component running on Core B could not starve the safety-related processes of needed CPU resource. For a more in-depth discussion of SMP and BMP, see Shiv Nagarajan's paper, "Processor Affinity or Bound Multiprocessing?".[8]

Preventing time starvation

Time partitioning helps ensure that all processes have access to sufficient CPU cycles to meet their time constraints. It separates CPU time into partitions, guaranteeing each process or group of processes a specific portion of CPU cycles, so that no process can starve other processes.

A specific form of time partitioning, called adaptive partitioning, can provide these guarantees while also ensuring that system resources aren't wasted. It assigns minimum levels of processor time to a group of threads *if the threads need it* (see Figure 7 below). The pre-set partition boundaries are enforced when the system is running to capacity. However, if a process in one partition can benefit from more CPU cycles, and processes in other partitions are not using their allocated time, the OS adapts the partition boundaries to lend the unused cycles to the process that can use them.

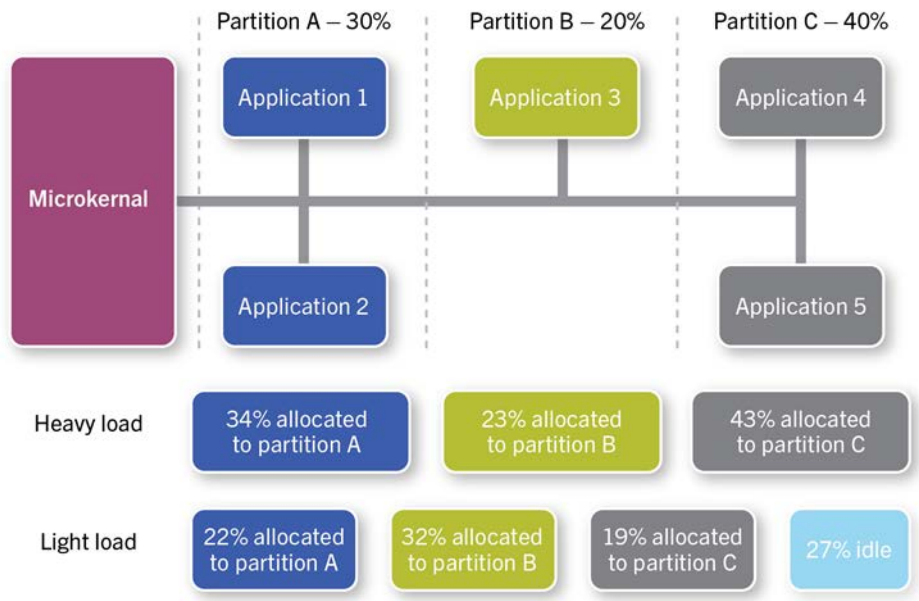
Partition A – 30%
Partition B – 20%
Partition C – 40%
Microkernal
Application 1
Application 3
Application 4
Application 2
Application 5
Heavy load
34% allocated to partition A
23% allocated to partition B
43% allocated to partition C
Light load
22% allocated to partition A
32% allocated to partition B
19% allocated to partition C
27% idle

Figure 7. An example of adaptive time partitioning.

In simpler systems, scheduling policies and tools such as rate- and deadline-monotonic scheduling can help ensure that processes meet their real-time deadlines. For example, with rate-monotonic scheduling, the processes with greater execution rates receive the highest priorities.[9] For some systems it is then possible to provide mathematical proofs that real-time deadlines will be met.

Preventing illegal memory access

A hardware MMU can prevent illegal memory access in conjunction with the OS. If the OS supports this feature, the MMU will prevent a process from reading, or writing to, the memory of another process. This should be a required feature of any software architecture used for a safety-related system.

Preventing data corruption

Protection against data corruption includes checksums, simple replication, data diversification, and sanity checks. Of all these options, a checksum or cyclical redundancy check (CRC) added to the data probably uses the least memory, but does not allow repair of corrupted data. Replication simply copies the data over to more than one location, sometimes remotely. With diversification, the same data is stored in more than one location, in different semantic ways. Sanity checks ensure that the data read is within acceptable parameters and rejects anomalies.

These techniques can all increase memory or CPU overhead. However, they can all be useful in systems that require guaranteed operation under adverse conditions, so the cost of implementing them should be considered during system design.

Preventing babbling

Babbling and malicious denial-of-service attacks can be contained by an anomaly detection program that learns what constitutes normal behavior and takes corrective action when the system begins to behave outside expected boundaries. Also helpful for this type of testing is “fuzzing,” where you intentionally call program functions with malformed inputs or in unexpected ways as a way to uncover conditions that the system cannot handle correctly.

Preventing deadlocks

Deadlocks occur when cooperating processes wait for each other to complete an action. Because both processes are waiting for the other to finish, progress stops.

Priority inheritance can help prevent deadlocks. It solves the problem of priority inversion, where a low-priority task prevents a task with a higher priority from completing its work. Priority inheritance prevents priority inversions by assigning the priority of a blocked higher-priority task to the lower-priority thread doing the blocking until the blocking task completes (see Figure 8 below).

Hardware watchdogs can also help address deadlocks, but they may not always detect a deadlock because they are oblivious to processes from which they are not expecting a “kick.” In comparison, a software watchdog, or high-availability manager, can help ensure that progress is being made. If the system architecture allows separate components to be stopped and restarted without a complete system refresh, a software watchdog can stop and reset the offending process or processes while the rest of the system continues to run.

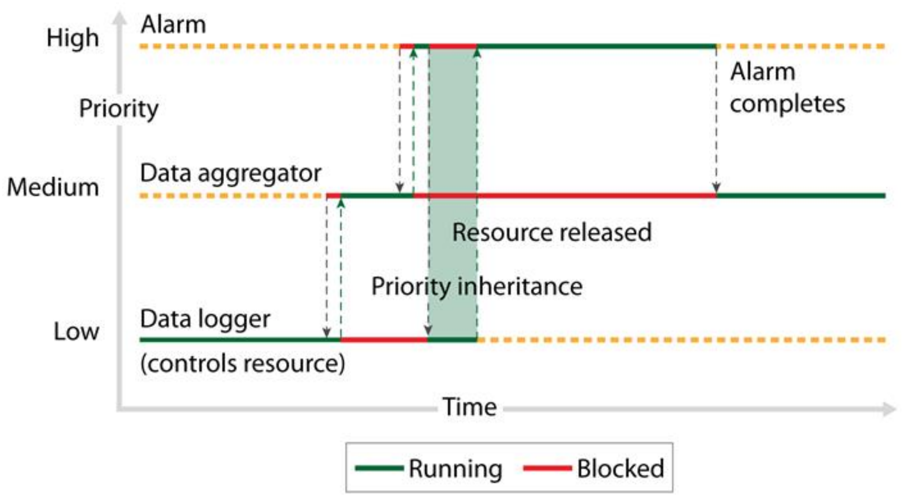

Figure 8. Priority inheritance can prevent lower-priority threads from blocking the execution of higher-priority threads.

Summary

In response to customer demands for more applications, features and services, and to economic pressures, automakers are consolidating non-safety-related and safety-related components on a single platform in their vehicles.

A microkernel OS can provide a full set of OS features to support consumer demands while ensuring that the system meets its safety requirements. The trusted code in a microkernel OS is simple and small, with a well-tested and short execution path that is granted system-level privileges. In short, a microkernel OS is inherently appropriate for safety-related systems.

When a system design calls for two different operating systems to co-exist on a single piece of hardware, a hypervisor is an effective solution. Nonetheless, our experience building both in-vehicle infotainment systems and safety-critical systems (including systems built with the IEC 61508 SIL3-certified QNX OS for Safety) suggests that a single microkernel OS can provide all the features needed for infotainment systems and the dependability and isolation guarantees required by ISO 26262.

Notes

[1] Roger Rivett, "The Challenge of Technological Change in the Automotive Industry", in C. Dale and T. Anderson (eds.), Achieving Systems Safety: Proceedings of the Twentieth Safety-Critical Systems Symposium, Bristol, UK, 7-9th February 2012, London: Springer-Verlag London Limited, 2012, p. 35.

[2] ISO 26262 Road vehicles—Functional safety, first edition, 2011.

[3] *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems (E/E/PE, or E/E/PES)*

[4] Tom Anderson, Letter to the editor, *Safety Systems*: 22-3, May 2013.

[5] ISO 26262-6, 15 November. 2011, p. vi.

[6] Reason, James. *Human Error*. Cambridge: Cambridge UP, 1990. See also Chris Hobbs, "Building Functional Safety into Complex Software Systems", Parts I and II, QNX, 2011.
< www.qnx.com/download/feature.html?programid=21862>
<www.qnx.com/download/feature.html?programid=21978>

[7] See also ISO 26262, Part 6, Annex D.

[8] Shiv Nagarajan, "Processor Affinity or Bound Multiprocessing? Easing the Migration to Embedded Multicore Processing", QNX, 2009
<www.qnx.com/download/feature.html?programid=20412>

[9] Briand, Loïc and Daniel Roy Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach, IEEE Computer Society, 3rd ed., 1999.