CHAPTER 6

Operating Systems in Multicore Platforms

Bryon Moyer
Technology Writer and Editor, EE Journal

Chapter Outline

Introduction

Embedded systems have a wide range of ways of providing low-level system resource and service support for software applications through the operating system (OS). They can have no support at all, or they can

Real World Multicore Embedded Systems.
DOI: http://dx.doi.org/10.1016/B978-0-12-416018-7.00006-7

provide rich system services of the order of what desktop computers provide, and even services not found on the desktop. One of the challenges of embedded systems is specifically that range: there's no standard way of configuring an OS for an embedded system. "Over-configuring" can introduce unnecessary complexity, overhead, and cost, while "under-configuring" can leave too much work for the application.

There are different high-level ways of approaching the problem, however, and they tie into some of the general architectural definitions mentioned in the Architecture chapter. In particular, they tie into the memory organization, since one of the defining aspects of the classic multicore architectures is how the operating system views and manages memory. The alternative configurations all represent different balances between the need for high performance and predictability on the one hand and the desire for some entity that can isolate an application program from low-level system resources and, to some extent, from other application programs on the other hand. As a general rule, more services and isolation tend to mean lower performance. Exactly what that trade-off means varies by system, and, as new architectures are devised, coupled with increasing processor and memory speed, the relative costs have diminished over time. But there are costs nonetheless.

We will discuss three fundamental ways of looking at a multicore system through the eyes of an OS: symmetric multiprocessing (SMP), asymmetric multiprocessing (AMP), and lightweight or bare-metal systems (the latter of which has its own chapter). For this discussion, we will assume "simple" or "plain" instances of OSes that directly access the hardware that hosts them. It's possible to add further abstraction through virtualization; that will be the subject of the following chapter.

It can be helpful to clarify a few concepts that become important when discussing OSes. We introduced the generic concept of "tasks" in the concurrency chapter. In this chapter, we need to be more specific. A "process" defines the scope of what we conveniently think of as "a program". From a practical standpoint, it circumscribes the namespace that a typical program can access without any special effort. It is possible to get two processes to talk to each other, but it takes extra work, and the compiler of one won't understand the names defined in the other. (Of course, it may be possible for one process to access the

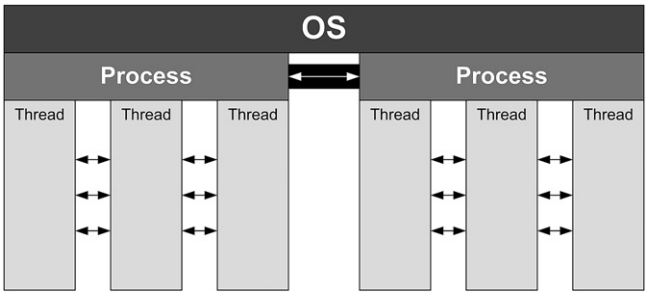

Figure 6.1
Process and threads. Threads can easily intercommunicate; inter-process communication must be explicitly built.

memory space of another through the magic of pointer arithmetic in the C language, but that's generally considered a bug – and a weakness of the C language.) Figure 6.1 illustrates this relationship.

A "thread", on the other hand, is part of a process. It's a path of execution that can conceptually proceed independently of another path. That possible parallelism remains conceptual in a single-core system, which offers no true concurrency. But in a multicore system, these threads can proceed at the same time – meaning that, at times, thread-safety issues that remained under the radar in a single-core machine may suddenly rise up to cause problems in a multicore system.

Even though threads have a conceptual level of autonomy, they are sub-units of a single process, so they understand the same namespace and share the same overall scope. Whether running on a single core, or on a multicore system, some operating systems support only one process, while others support multiple processes. Those that support one process simply manage a set of threads within a unified address space. Those that manage multiple processes manage multiple address spaces, and multiple threads within each such space.

When discussing high-level concurrency issues, we tend to refer to both processes and threads generically as "tasks". But when discussing OSes and, in particular, scheduling, they are often also referred to as "threads," or "contexts". When an OS decides to move one task or thread out of execution and swap another one in, a "context switch" has occurred. That

context might reflect a different thread or task from the same process as the one that got swapped out, or it might reflect a different process altogether. At this level, it doesn't matter.

Symmetric multiprocessing systems and scheduling

SMP (Figure 6.2) is the easiest configuration from the standpoint of a programmer. In an SMP architecture, all processors are identical (homogeneous), and all can access a common memory. They may also have access to "private" memory as well, but what makes a system SMP is homogeneous processors with access to a common memory. Since all processors are identical, they can all execute the same code from memory, even at the same time. This enables code to be run on any processor at any time, providing an attractive parallelism and transparency to the programmer. To the programmer, an SMP system effectively makes many cores look like a single core – although it doesn't hide concurrency issues. The SMP OS, on the other hand, must manage the use of each core at all times, and this scheduling operation can be completely hidden from the application, which might have no explicit knowledge or control over which thread is running on which core at any time.

An OS that can handle multi-threading on a single-core system needs to have a scheduling facility in place, so the move to SMP does not introduce the need for a scheduler. What changes is the level of complexity: instead of scheduling threads and processes on a single core, it must do so across

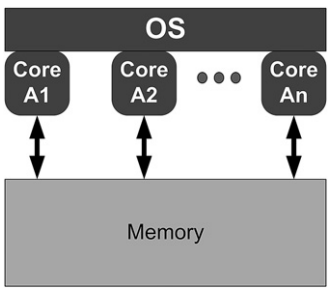

Figure 6.2
SMP configuration. All cores are identical and see the same shared memory.

several cores, and how it does that can have an enormous impact on system performance. Likewise, an application composed of multiple processes or threads that has been developed to run on a single core may be able to run on a multicore system without need for re-structuring. At the same time, re-structuring may be desirable to avoid concurrency issues, and/or to achieve desired levels of performance.

Real-time systems generally try to achieve responsiveness and throughput, in some application-dependent balance. Some applications are primarily concerned with responsiveness, and throughput is not the issue. Others may be designed to achieve high throughput, and responsiveness is consciously compromised to achieve higher throughput. The most fundamental challenge is balancing the desire to achieve higher throughput by keeping all cores busy against the trap of introducing unproductive operations that occupy cores – keeping them busy – without achieving productive results. This impacts many properties of a scheduler, but paramount among them is controlling the "affinity" between a given thread and a given core. The OS can allow full migration of threads across all available cores (no affinity), or force certain threads to run on certain cores (full affinity), both methodologies having benefits under certain application-dependent circumstances.

Generally, in multi-threaded systems, a programmer assigns each thread a priority that determines which of several threads that might be "READY" to run (not blocked or waiting for some external event) is actually given the opportunity to run. In a multicore system, this situation extends to each core. In a priority-based, preemptive scheduling system, the highest-priority thread that is READY to run is the one that will be allowed to run on a given core. But adherence to a strict priority scheduling algorithm might result in excessive context switches, and introduce significantly undesirable overhead.

The frequency of shifting may depend on overt policy or may be more subtly determined by system parameters. Specific apportionment is possible through policies like "round robin" or "weighted round robin" scheduling algorithms, where each thread or process gets a designated slice in turn. Such policies tend to be used more in real-time operating systems (RTOSes) due to their focus on deterministic program behavior.

Absent such policies, scheduling decisions boil down to the fundamental question of how long a stall merits a context switch, and how high a priority thread is waiting, READY to run. Switching contexts (other than the simple need to give each context a chance to proceed if there are more contexts than cores) is both about enabling important threads to have a chance to do their job and avoiding dead time in the processor if other work awaits. So if one thread initiates an action that will take some time to complete, without that thread's direct involvement (such as an I/O to a device), then, instead of waiting around for the device to complete its operation, another thread's context can be swapped in and given the use of the processor in the intervening time.

But it takes time to swap contexts — and how much time it takes depends on the nature of the system. Each context has a state that must be preserved. On a simple core, that state is located in memory, so swapping contexts means moving data into and out of memory — which can take a nominal amount of time (and will presumably make use of fast local memory). On a core with hardware hyper- or multi-threading, however, provisions have already been laid for multiple contexts, so little data movement is necessary, and a switch between two contexts which both have hardware resources can happen very quickly. Again, if more contexts than dedicated hardware resources exist, then context switch overhead becomes an issue.

Moving threads from one core to another, however, is an entirely different matter. In an SMP system, it doesn't really matter where any single thread executes, which means it could, in theory, stall on one core and then, if it's having a hard time getting going again, be moved to another core that's not so busy. That makes for more comprehensive, but not necessarily more efficient, core use. This is due to the fact that, as illustrated in the memory chapter, the context has to move, and the cache on the new core has to be refilled with the useful things that were already in place on the old core.

All of this takes time, so, while the program should perform correctly at a functional level regardless of where the threads execute, performance suffers if threads are moved around too much. The caching structure may also make it desirable to favor one core (say, one that shares an L2 cache) over another if a context has to be moved. This illustrates an

example of when directing the OS to restrict a thread to a subset of cores might pay dividends.

This becomes part of the heuristic nature of scheduling algorithms, and it is why, in situations where a designer knows very specifically how threads and processes will behave in a dedicated program, he or she might benefit from a scheduler that allows the user to "tune" its performance in light of these considerations, or even to replace a stock scheduler with a custom one that incorporates that knowledge. Customizing the scheduler, however, while possible, is not common. Rather than taking that drastic a step, it's usually possible to guide the OS in scheduling operations; this will be discussed below.

The SMP configuration is widely supported by operating system providers, both in large-scale and real-time OSes. The larger OSes, of which Linux is most prevalent, tend to provide a richer array of services and options. The better-known options and providers as of writing are shown in Table 6.1.

Applications requiring a higher degree of determinism and/or a smaller footprint or lower service overhead than is provided by large-scale OSes will turn to RTOSes instead, and many of them have provisions for multicore. They're illustrated in Table 6.2.

There is some debate as to whether Linux has a true real-time configuration. One company provides a Linux package that they claim to be suitable for real-time use. Opinions vary; in general, it may indeed work well for a number of different applications, but the challenge comes for systems demanding hard-real-time that can be proven through

Table 6.1: Large-Scale OS Support for SMP

OS	Provider
Linux	Wind River (Intel)
	MontaVista (Cavium)
	Mentor Graphics
	Enea
Windows Embedded	Microsoft
Android	Wind River (Intel)
	Mentor Graphics

Table 6.2: RTOS Support for SMP

OS	Provider
DDC-I	DEOS
INTEGRITY	Green Hills
Lynux Works	LynxOS
Neutrino	QMP
Nucleus	Mentor Graphics
OSE	Enea
QNX	Momentics
ThreadX	Express Logic
VxWorks	Wind River (Intel)

calculations. Linux is a complex OS, and it's doubtful that all of the possible paths of execution could be traced to guarantee a worst-case latency. This remains an open question.

While there are many benefits to the SMP model from the standpoint of the programmer who doesn't wish to delve too deeply into the complexities of concurrency, it has its limitations. Most significantly, it doesn't scale. As discussed in the memory chapter, if too many cores try to access the same memory, at some point the bus chokes on the traffic, and no further performance improvement is possible regardless of the number of cores added – in fact, performance can decrease due to bus thrashing.

Having a single shared memory space can also leave one thread or process open to corruption by another thread or process. Separation between spaces is virtual, not physical, so errant pointer arithmetic, for example, can easily play havoc with memory that ostensibly belongs to a different context. While virtualization and "trusted zone" kinds of architectures provide the strongest solution for this, it is possible to improve things without them.

And finally, heterogeneous and NUMA architectures cannot take advantage of SMP since they violate the assumptions of "all cores are alike" and the single shared memory block.

These OSes typically come with a set of tools to help develop the system and diagnose issues. Those tools are described in more detail in the Tools chapter later in this book.

Assymetric multiprocessor systems

An AMP system is somewhat harder to define than an SMP system is. In essence, if it's multicore and it's not SMP, then it's AMP. AMP systems may have a number of different looks. We'll look at some common examples, but these by no means define what AMP is. In fact, there are different views on what constitutes AMP vs. SMP. One view is that these are hardware distinctions, with SMP being synonymous with a homogeneous set of cores with an UMA memory configuration; anything else is AMP. The challenge is that a homogeneous/UMA system can be configured in a way that's not SMP, by running a separate OS instance on different cores (per the following section). The other view includes the OS in the picture, combining that with the hardware to define SMP and AMP. According to that view, what makes a system AMP is the fact that there is more than one OS instance in the system.

OS-per-core

The simplest AMP configuration is one where each core has its own independent instance of an OS (Figure 6.3). If the set of cores is homogeneous with a shared physical memory, then it looks similar to SMP, except that all cores do not necessarily have access to the same shared memory. If you prefer the hardware-only definitions of SMP/AMP, then this specific case is the "exception": it's SMP configured as AMP. Note that separate virtual memory spaces may be separate regions within one physical memory, or they may be different physical memories.

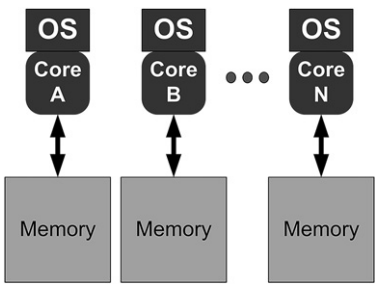

Figure 6.3
AMP configuration. Each core has a separate OS and memory.

From a software standpoint, the big difference between AMP and SMP is the fact that each core in an AMP system has separate processes; no process can be distributed over multiple cores in the way that SMP allows, since the cores may be of different architectures and execute a different set of instructions. If a multi-threaded process is moving from an SMP to an AMP architecture, it means that the original monolithic program has to be split into multiple independent programs, each one compiled and linked into an executable for the specific architecture of the core on which it will run. The cut lines may or may not coincide with the thread boundaries in the original program.

This exposes the fact that a process has a common namespace. A global variable is no longer accessible from all cores; it is only understood by the OS hosting the code where it's defined. Of course, you would never get to the point of an actual running program failing to find that global variable – the independent programs would never compile correctly in the first place. The main point is that distributing a program over multiple cores in an AMP setup requires a fair bit of work just to get things to run again. Whether they run effectively or not depends on the partitioning chosen, which is a topic covered in its own chapter.

Applications that are naturally segregated make ideal AMP candidates, where each type of processing in the program can be configured to run on a processor that excels at that type of processing. Further, it can run under an OS that is also ideal for that type of processing. This can make for very efficient systems where hardware and software are tailored to fit the individual needs of separate aspects of a program. A common configuration of this type is one incorporating a RISC CPU and a DSP on an SoC, with the RISC processor handling general application tasks, perhaps running Linux, while the DSP, perhaps running an RTOS, handles the computationally demanding signal processing or math functions somewhat independently. The two subsystems require only minimal interaction and synchronization, enabling each to run efficiently doing what it does best.

Other partitioning considerations aside, it's usually impossible to pull apart a program that was not designed to run on an AMP system in a way that completely isolates variables in one process. If the original program

has data that is widely accessed by different portions of the program, then, after splitting the program apart, those separate entities are still going to need to access shared data. This requires implementation of inter-process communication (IPC), a topic covered in the synchronization libraries chapter. It's likely that you would want to minimize the amount of such communication, however, as it entails software (and potentially hardware) overhead. The partitioning process effectively cuts lines of communication, and, collectively, they constitute cutsets along the boundaries. As a general rule, you want to minimize the size of those cutsets.

Ideally, a system would be designed from the start for operation on a specific type of AMP system, and IPC considerations could then be factored into the program structure without risk of creating a web of access that would be difficult to unravel into a segregated system after the fact.

Figure 6.4 provides an illustration of this point. The program shown is split up into four partitions that intercommunicate.

If these partitions are to be split amongst two cores, then there are numerous ways to group them. As will be made clearer in the Partitioning chapter, part of the consideration is the amount of computing in each core; that should ideally be balanced, from the perspective of achieving the minimal maximum execution time for any core. But reducing communication is important as well; from that

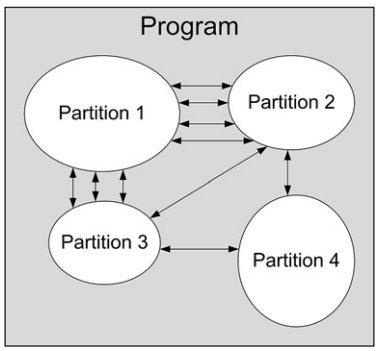

Figure 6.4
A program split into four natural partitions that intercommunicate.

standpoint, Figure 6.6 is preferable to Figure 6.5, since the latter cuts six lines of communication, while the former cuts only two.

Note that these figures are merely intended to illustrate the overall concept. In a real partitioning scenario, it's not just how many lines of communication there are, but how much data needs to be transferred, and how often, that also impact the decision. Such details are covered in the Partitioning chapter. The point is to give careful consideration to the communication, as well as computational needs of the program, and if possible tailor the hardware resources (processors and memory) to suit the program's needs. Whether six lightly used communication channels are

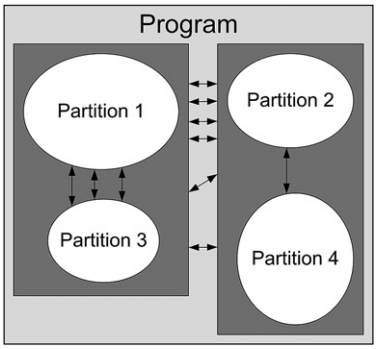

Figure 6.5
The partitions grouped in a way that requires six lines of communication.

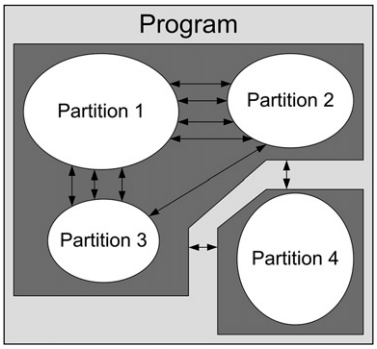

Figure 6.6
The same program split in a way that requires only two lines of communication.

preferable to two heavily used channels depends entirely on the nature of the application, and neither one is inherently superior to the other.

For these reasons, neither moving from SMP to AMP nor developing for AMP in the first place is to be done lightly, as it has fundamental implications for the structure and implementation of the program. Programs originally conceived for AMP may include things like cellphones, with a heterogeneous architecture (e.g., RISC plus DSP) requiring a separate OS (if any) per core, and pipeline architectures – although the latter are often implemented in something closer to a bare-metal configuration, to be covered below (and in its own chapter).

Multiple SMP

Another AMP implementation keeps SMP in the picture, except that, instead of all the cores being under the dominion of a single SMP instance, the cores are partitioned into one or more groups, each of which gets an SMP instance (Figure 6.7). This may also be referred to as *hybrid SMP-AMP*.

This scheme provides one way of more easily controlling access to resources between programs while still taking advantage of the benefits of an SMP environment. In an extreme instance of such an architecture, if an SMP system were reduced to one core, then the result would be an AMP system. Conversely, any AMP system can be generalized as being a collection of *n*-core SMP subsystems.

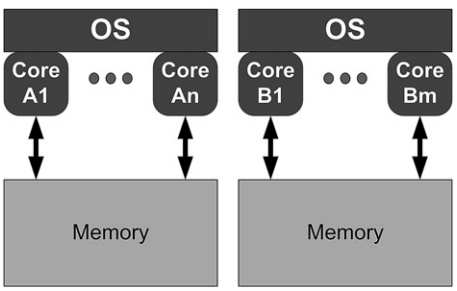

Figure 6.7
Hybrid SMP-AMP configuration.

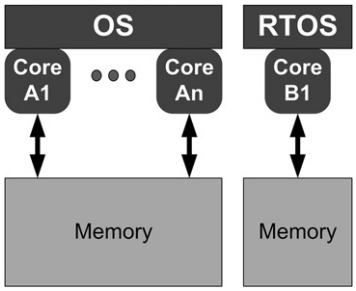

Figure 6.8
An SMP configuration with an RTOS on one core.

SMP + RTOS

Another very common practice, briefly mentioned earlier, couples the simplicity of a heavyweight OS with the deterministic nature of an RTOS (Figure 6.8). The idea is to quarantine the time-critical elements to operate under the RTOS while giving the non-time-critical portions the services of a rich OS. If it's just a two-core system, then each core gets a single-core instance of its OS. But either the full OS side or the RTOS side or both could have more than one core, in which case SMP instances could be used. As noted, such configurations might use Linux for the OS and an RTOS alongside for the demanding real-time activities.

SMP + bare-metal

Some functions are so performance-critical that they can't tolerate any system overhead. In those cases, designers try to operate with no OS at all. That's a lot of work, and often counter-productive. For example, operating with no OS requires the application to manage all the activities of the program within its own code. That code is not likely to operate as efficiently as a fine-tuned RTOS scheduler, resulting in additional overhead.

Some providers like Cavium and MontaVista (now part of Cavium) offer thin "executives" that provide a bare minimum of services with a focus on efficiency. As mentioned, this is a topic of a separate chapter, but such setups are often combined with at least one other core (often referred to as the "host") that's running a full-up OS like Linux. This is

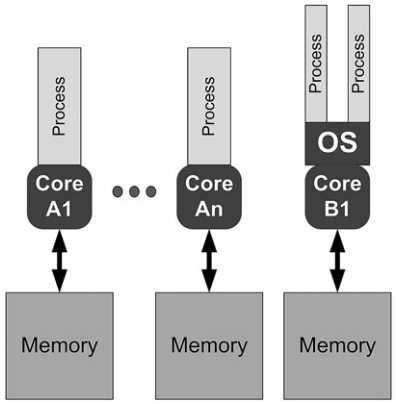

Figure 6.9
A set of bare-metal cores plus a "host" core running a full-service OS.

yet another AMP configuration (Figure 6.9, with multiple bare-metal cores). In this case, the "thin executive" is simply a type of RTOS, perhaps proprietary, and perhaps tuned to a specific architecture.

This is most common in packet-processing applications. The bare-metal cores are optimized for speed, constituting the "fast path" of the "data plane" for the most common packets that need to be handled as quickly as possible. The core handles both "control plane" activities and the "slow path" of the data plane. The cores may be identical, or, as in the example of Intel's IXP network processors, use micro-engines for the fast path and a larger ARM XScale core for the host. The IBM/Sony "BE" incorporates one central core and eight peripheral cores that generally run a small RTOS and small network stack, while the central core runs Linux.

Many OSes claim "support for AMP". Yet when used in a non-SMP setup, they're simple single-core OS instances. So what does "AMP support" mean, since the OS in such a mode knows nothing of multiple cores?

The answer boils down to initialization: the system needs an organized way to boot up, and this typically means that one core is considered the master; it coordinates bring-up of the entire system. So AMP support more or less reflects the availability of a board-support package (BSP) that can initialize an AMP system. We'll discuss more about boot-up later in this chapter.

Virtualization

AMP setups use multiple OS instances, but the cores define the boundaries of those instances. One might, however, want to share the resources of a given core between two OSes. Or you might want to have some entity outside the individual OSes that manages shared resources like I/O amongst all of the OSes, none of whom know anything about the existence of the others. Or you might want to strengthen the isolation between processes and cores and resources to make the system more robust and secure.

This is the realm of virtualization, which explores new architectures and ways of layering services and programs. Virtualization is the topic of the next chapter.

Controlling OS behavior

Various operating systems will give different levels of control of their scheduling and other operations to the user. Different applications will require different levels of control. Processes that aren't performance-critical running under an SMP OS are likely to let the OS do what it thinks is best. By contrast, in a safety-critical application with hard real-time requirements, you need to know and control how the thing will run in as much detail as possible.

OS behavior can be controlled in a number of different ways, and the following sections describe some of the more common ones.

Controlling the assignment of threads in an SMP system

As mentioned, SMP scheduling algorithms can be complex, using priority, fairness, time-slicing, and "affinity" — in one sense — to determine what to schedule where. Affinity in this context refers to the general desire to keep a suspended thread on its original core, since moving it means you have to reload a new cache, which hurts performance.

One way of taking control is to assign threads to cores manually. An SMP OS may offer the application the ability to decide for itself when to restrict certain threads to certain cores, and conversely, when to prevent

certain threads from running on certain cores. You might assign every thread, or you might merely assign critical threads to, say, one core, leaving the OS to assign the other threads to the remaining cores as it sees fit. This process of assigning threads is variously referred to as "pinning," "binding," or "setting thread affinity".

Threads can be assigned to cores statically by means of the start-up configuration file, or they can be assigned when they're created, or they can be assigned in real time using an OS call. Every OS has some facility for managing this, although the details may vary between OSes.

For a given thread, you may be able to assign it to a specific core, or you may be able to give it a bit mask that gives the OS flexibility in assigning it to one of a few specific cores that you have specified. Conversely, the OS (or RTOS) may enable the developer to restrict the thread from running on certain cores, enabling those cores to be dedicated to running certain other threads and only those threads. Note that, for some OSes, the assignment is sometimes treated more as a "suggestion" – the OS tries to honor your assignment, but may not. With RTOSes, the assignments are guaranteed.

For example, the OS may define a bitmap of cores within a 32-bit word, or multiple words for larger systems. Each bit represents a particular core in the system. Then, for each thread, a bitmap specifies which cores that thread is allowed to run on and which it cannot run on. By using such a mechanism, threads can be "locked" to a single core, while other threads are allowed to share remaining cores. In systems where there are more threads than cores, then "multi-threading" strategies also come into play. In those cases, several threads must share a core, and context switching is a necessity if all threads are to run. Context switching must be highly efficient, lest more overhead gets introduced than is saved by parallel execution!

Controlling where interrupt handlers run

Interrupt handlers can also be pinned, although it is much more typical that this would be done statically, at boot-up. It's much less common for these to be moved in real time.

The flexibility you may have for pinning and interrupt handling depends very much on the system. Some multicore systems dedicate specific interrupt lines to specific cores, in which case you can't move them (unless you use indirection to "bounce" the handling from one core to another, which causes additional latency). For example, on some ARM cores – particularly those using their TrustZone architecture (covered more in the next chapter) – fixed/fast interrupts are hard-wired to core 0.

How you actually make the interrupt assignment varies by system. It's typically an API call, BSP that defines the board configuration, and it might even require some assembly-language code changes to low-level initialization runtimes to implement.

For systems where multiple devices share a single interrupt line, note that reassigning the core that will receive the interrupts will move all of the devices together; you can't move only some of the devices without reassigning the physical connections to different interrupt lines.

Partitions, containers, and zones

Some OSes have a notion of combining certain processes together for isolation purposes. Green Hills and QNX implement what they call *partitions*; Linux has a *containers* feature, which is similar to what Solaris and BSD/OS refer to as *zones*.

Partitions allow the grouping of processes into a partition. Only one partition can be active at a time, meaning that processes in different partitions can never interact. The benefit is better predictability for safety-critical applications. It comes at the cost of reduced concurrency, since, by definition, only some of the processes can be executing concurrently.

On some newer OS versions, this concurrency limitation is relaxed to some extent by allowing one partition to be assigned to one set of cores and another partition to a different set of cores. In this manner, more things can be running at the same time, while maintaining the isolation between partitions. Communication among partitions is carefully managed through secure services of the OS, rather than an open region

of memory that might become corrupted for one partition due to the actions of another – a fatal flaw in the separation of partition requirement.

Containers or zones act like OSes within OSes, allowing processes in a container to have their own scheduler and dedicated access to resources. This is sometimes referred to as "OS virtualization", although it's not true virtualization.

A broader discussion of OS separation and virtualization are provided in the following chapter.

Priority

Priority is a critical parameter associated with threads and interrupts; it has a critical impact on how the scheduler decides what tasks or threads to run when. In multicore systems, it works pretty much the same way as it does in single-core systems. In fact, you could say that multicore alleviates, to some extent, conflicts between tasks having the same priority, since there's some chance that they can both run concurrently on different cores.

However, many single-core systems that operate with priority-based preemptive scheduling, which includes most RTOSes, enable developers to become (consciously or subconsciously) dependent on the principle that, in a multi threaded system on one core, a task or thread can rely on the fact that no lower or equal priority thread is running at the same time. Running such an application on a multicore system, under an OS that allows threads to run when "READY", can lead to catastrophic results that are hard to debug.

Some RTOSes offer the user the means to run the SMP system in a "single-threaded" mode, where priority order is maintained, or to consciously relax those restrictions and enable threads of different priority to run on different cores at the same time. Presumably, using this mode requires the user to be confident that such a mode of operation will not introduce any race conditions or priority conflicts.

Priority can be set statically or on the fly. Some systems allow a weighting in addition to the priority so that the scheduler can be told to

give more time to specific tasks in a "time-sliced" mode. Some systems also have the concept of a "maximum" priority: in addition to the "current" priority, a maximum can be statically set. Then, if the priority of that task is raised, it can never be raised above the maximum, and the task can't spawn threads with priority higher than the maximum.

Another priority variation found in Express Logic's ThreadX RTOS is "preemption-threshold", which optionally sets a separate, higher, priority threshold that any preempting thread must satisfy. These scheduling policies and services enable the developer to achieve desired results under application-specific operational situations.

Kernel modifications, drivers, and thread safety

One of the thorniest issues bedeviling systems ever since the concept of multi-threading came along is that of thread safety, especially when monkeying with low-level OS code and drivers. For operating systems that have a kernel mode and user mode, most programming is done in user mode. That keeps the OS intact, and if the program crashes, it only affects the individual program, not the entire OS.

But, historically, when customizing or building critical low-level capabilities for embedded systems, it's been common to add or patch modules within the kernel both for better performance and, essentially, to make it look like the OS has new or modified services. These so-called "kernel mods" can be extremely tricky to debug, and failures can bring down the entire system.

There are different views at present as to what the best approach is here. One is simply that there is no longer a real performance issue to working in user mode, and that all programming should be done there. Another approach moves things from a monolithic OS to more of a "microkernel" arrangement; this will be discussed in the next chapter.

Thread safety has always been a critical determiner of the quality of a driver, and that's no more or less true with multicore than it is with single-core multi-threaded. Of course, all code must be thread-safe, but drivers can wreak more havoc than application code if they fail. The issues that lead to unsafe threading are the same for multicore as for

single-core, except that the risks are higher due the fact that code can truly be concurrent.

In addition, some of the "fixes" used for single-core systems won't work for multicore systems. One of the more common threats to thread safety is the unpredictable firing of an interrupt while some critical region of code, which is intended to run uninterrupted (Figure 6.10a), is executing. The interrupt handler will start running, bumping whatever task was in progress regardless of where it was (Figure 6.10b). If the critical region and the ISR that interrupts it modify and read the same memory locations, the system could end up in an inconsistent state.

The easy solution for single-core systems is to disable interrupts while executing critical sections of a program. That way you know you'll get through the delicate parts with no interruptions; you re-enable interrupts after. But this won't work on a multicore system. When you disable interrupts, it only affects the one core. The interrupt may fire on a different core. Rather than interrupting the critical code that's running on one core, the interrupt handler will actually run concurrently on another core (Figure 6.10c). As a result, the mischief it can create may be different from what would happen in a single-core system, but it will be mischief nonetheless.

Some RTOSes address this issue through implementation of a policy that disables the scheduler, as well as all interrupt processing, when working in a critical section. Protection of critical sections from interrupt processing is expanded to protect against multi processor access. This protection is accomplished via a combination of a global test-set flag and

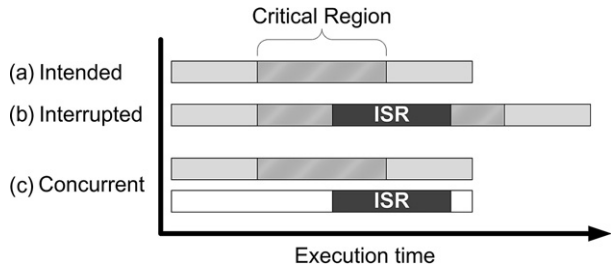

Figure 6.10
(a) Uninterrupted critical region; (b) critical region interrupted on single core; (c) ISR running concurrently with critical region.

interrupt control for the core holding the test-set flag. This solves the problem, but introduces potentially significant overhead by "freezing" all other cores while a critical section is being modified by one core.

With multicore systems, the OS must take care to protect the resources whose integrity is critical. Programmers can take control of some of this protection by using locks (or lock-free programming techniques) within the application itself. Different kinds of locks, as well as lock-free approaches, are discussed in the synchronization chapters that follow.

System start-up

One of the things that distinguish embedded systems from their general-computing relatives is the wide variety of configurations. In particular, peripherals and I/O can be dramatically different between systems. The OS has to know what's in the system, where to find it, and what code to run in order to exercise the different peripherals.

This holds true for single-core and multicore systems. But with multicore systems, you have one more element to deal with: getting the cores themselves in synch. As built, all the cores are usually the same; no one core is any more important than any other one. But someone has to be in charge of bringing the system up, so one core is allocated the role of "master", at least for the purposes of boot-up.

During system initialization, there are a variety of tasks that must be performed on a system-wide basis; each core may also have some initialization work to do that only affects the individual core. This means that each core has initialization code to run, but the master core will have more to do, since it's in charge of configuring the global portions of the system.

This is made possible through the use of "barriers". A barrier is a software construct that establishes a kind of meeting point or rendezvous. When any core reaches that part of the code, it holds there. When all of the cores have arrived at the barrier, then the barrier is lifted, and they can all move forward.

How this plays out during boot-up is that all the cores can start executing their initialization code, but the master will take longer since it has more

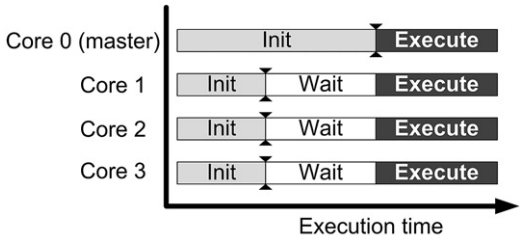

Figure 6.11
Four cores booting up; when all four have reached the barrier, then execution can begin.

to do. So there is a barrier at the end of each initialization routine, and, once everyone has finished – and, in particular, the master has put the system into a stable starting state, then the barrier is released and the cores can start doing their work. This is illustrated in Figure 6.11.

You'll see a specific example of initialization code in the bare-metal chapter towards the end of the book.

All of the information the system needs to boot up is typically put into a configuration file that is accessed during start-up. This is a critical part of the BSP for a given system. For the most part, these files and their formats are different for each system or OS being used.

An example of such a file can be found in one standard that was set by the Power.org group; it's called ePAPR [1]. The contents include a device tree, which defines the topology of all of the chips and other "devices" on the board. It also specifies the image formats for "client" programs as well as their requirements: identifying the master CPU, specifying the entry point for the CPUs, and various other requirements and considerations for both SMP and AMP configurations. Details on device bindings and virtualization are also included. All in all, files like these provide the OS with all of the information it needs to manage the system.

Debugging a multicore system

Fundamental to the processing advantages of a multicore system is the unfortunate complexity of the software that must run on it. Both the OS and the application are subject to challenges introduced by the fact that

code is being executed on multiple processors at the same time. From a software perspective, this introduces several complications to the debugging process, not the least of which is the difficulty of determining just what the system is doing at any point in time.

In a single-core system, a breakpoint can be used to pause the system at any instruction (high-level code or assembly), and at that point, all system resources can be examined: registers, memory, application data structures, thread status, and the like. This can paint a clear picture of the system's status, and the developer can use this information to determine whether the system is performing correctly or not, and also to enable it to perform better.

But in a multicore system, the picture on one core must be correlated with what is going on with all other cores, and this can be challenging. Breakpoints alone do not do a very good job of painting such a clearer picture. Fortunately, there are tools, as shown in Table 6.3, that help a great deal in unraveling the otherwise confusing array of system activities, and painting a clear picture of what the system is doing at any point in time. Operating systems such as Linux and RTOSes such as VxWorks, INTEGRITY, and ThreadX provide such tools, and they all operate in a similar manner (Table 6.3). Of course, each is unique, and features vary from one to another, but for the purposes of this brief discussion, they are sufficiently similar to discuss their common technology generically.

Table 6.3: Example Trace Tools for Debugging

Operating system	Company	Trace Tool
Linux	FSF	LTT
VxWorks	Wind River	WindView
INTEGRITY	Green Hills	EventAnalyzer
ThreadX	Express Logic	TraceX

The information gathered

In a hardware trace system, such as a Logic Analyzer, Emulator, or a debugger that captures information from ARM's Embedded Trace Macrocell, a stream of instructions is generated and can be captured, saved, and analyzed. A software tool, analogous to such a program

trace, is what is generically called an "event trace". An event trace can capture system and application information that is helpful in understanding what the software is doing in real time. A software "event" is generally any system service, state transition, interrupt, context switch, or any other activity the application would like to capture upon its occurrence. It might be when a particular value exceeds a certain threshold, for example.

While the software events occur in real time and can be represented at points in real time, they typically are gathered for post-mortem analysis. Some implementations do offer real-time display of these events, but they generally happen too rapidly for human interpretation as they occur. A post-mortem analysis enables the developer to delve into the details that occur over the course of a few milliseconds in order to discover exactly what has occurred.

The operating system is responsible for capturing and saving relevant information about each event. This information, for example, might include the type of event (e.g., message sent to a queue), which thread sent it (e.g., thread_1), when was it sent (e.g., timestamp from the system real-time clock), and other related information. The OS will save this information in a circular buffer on the target, so it will always contain information from the last "*n*" events, depending on how large an area of memory has been allocated for the buffer. Then, either as the system is running, at a breakpoint, or system termination (crash), the buffer contents can be uploaded to the host and displayed.

Uploading the information

In order to upload the captured event information from the target to the host, a variety of mechanisms can be employed. If the data is to be uploaded as the system is running, an Ethernet or USB connection might be used. If uploaded at a breakpoint, the JTAG debugger can be used to read the buffer's memory. Most debuggers have the capability to upload a memory region to a host file. By using this ability, the trace buffer can be loaded onto the host as a file, readable by the trace tool itself. Once on the host, then the events can be read, interpreted, and displayed graphically to show the developer what has been going on.

Painting the picture

The event trace tool runs on the host, reading the uploaded event buffer and generating a graphical or textual display of the information it contained. The buffer will contain a data "packet" for each event that has been recorded by the target OS. By digesting all of the packets, the event viewer can determine the exact time span that has been captured, and every event that occurred during that span. It also can associate each event with the thread that initiated the event; and can produce histograms of common events, an execution profile for each thread, each core, and system activities such as interrupt servicing. A timeline of events can be constructed, very similar to what is shown on a logic analyzer for signals over a period of time. Figure 6.12 shows an example of how this might look for a period of time in a two-core SMP system.

In this example, application threads are listed on the left, and this list is repeated for each core in the system – in this case, two cores. Across the top is a time scale of ticks, where each tick represents a period of real time that is dependent on the resolution of the clock source used. In this example, notice that the "Sequential View" tab has been selected, and the tick marks are simply a sequential counter, not

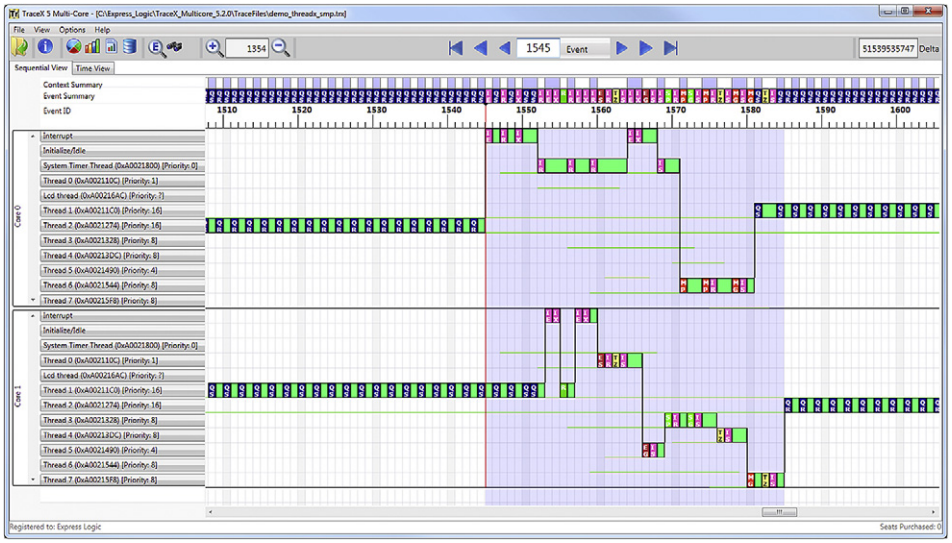

Figure 6.12
An event trace example.

correlated to real time. The correlation with real time could be seen in the "Time View" tab.

To the right of each thread, ordered from left to right, are the events that that thread has initiated, each depicted with an icon and a symbol to identify it. Vertical black lines indicate context switches between threads, and horizontal green lines indicate that the thread was running, but not initiating any events during that period. Note, for example, that thread_1 sends messages to a queue ("Queue Send" or "QS" event), and thread_2 receives those messages from the same queue ("Queue Receive" or "QR" event). Also shown are timer interrupts and activity from threads that are activated by the timer expiration (e.g., thread_0 on Core-1, and thread_6 on Core-0).

By examining the events for each thread and the events on each core, a much clearer picture of system activity can be seen than without such a tool. The developer can then determine whether system behavior is as designed, or if something has gone wrong. Further, by understanding the exact sequence of events, errors in logic or race conditions can be detected and corrected more easily.

Debugging a multicore system is indeed challenging, but the reward is achievement of greater performance. By utilizing all available tools for development and debugging, it is possible to simplify the challenge and shorten development time, while achieving desired levels of system performance.

Summary

The operating system can play a critical role in the operation of a multicore system. There are numerous ways of configuring one or more – or no – operating systems, and the best arrangement will depend on the application.

An SMP configuration has the most "hands-off" potential, where the OS scheduler can take care of making sure all tasks are executed. There are mechanisms available, such as the ability to bind processes or threads to specific cores, that allow some control over operation and the allocation of resources.

An AMP configuration allows much more flexibility and provides much more control, but it also means that everything must be managed explicitly. The interactions between cores, processes, and threads may not proceed transparently. But for applications requiring critical timing, the ultimate in performance, or guaranteed safety, the various AMP arrangements generally provide a better solution than SMP. AMP systems can also scale further than SMP systems can.

SMP is best suited to applications where the load is very dynamic – so much so that the application cannot easily keep all cores utilized in an AMP mode. AMP gives the best potential for performance if the load is more statically defined.

Virtualization adds yet more possibilities while addressing some of the vulnerabilities of systems running traditional operating systems. This is the topic of the next chapter.

Reference

[1] "Power.org™ Standard for Embedded Power Architecture™ Platform Requirements (ePAPR)", Version 1.1. <https://www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.1.pdf>.