By Paul N. Leroux,
Technology Analyst,
QNX Software Systems Ltd.

The new business imperative: achieving shorter development cycles while improving product quality

Most embedded software developers share a common assumption: productivity depends more on your tools than on your realtime operating system (RTOS). But even with the best tools, developers can spend significantly more effort on verification and maintenance than on creating new and innovative features for their products. The underlying problem? OS architecture. By their very nature, traditional OS architectures provide either no support, or limited support, for the memory management unit (MMU). As a result, an embedded software product can require extensive debugging and testing for even minor code changes. In this paper, we compare OS architectures and see how one approach, Universal Process Model (UPM™) architecture, helps developers redirect their efforts away from software integration and maintenance and back toward product innovation. Unlike other architectures, UPM allows applications, drivers, protocol stacks, and even OS modules to run in their own memory-protected address spaces. This approach not only eliminates needless debugging and testing, but also boosts availability at run time. For example, a UPM-based system can recover from software faults without rebooting and can support hot-swapping of both hardware and software, including most of the OS itself.

ARE TOOLS REALLY THE KEY?

All embedded operating systems (OSs) are the same, right? If you want to reduce time-to-market, make your product more reliable, and develop ”killer” features, your choice of tools will make a real difference - but your choice of OS probably won't.

Do you believe that? If so, then ask yourself these questions:

- Do your release dates keep getting further apart, even as you spend more effort on system development?
- Do you manage to release new products on time, but only by compromising on features or reliability?
- Are your QA costs rising exponentially as your products grow more complex?

If you answer yes to any of these questions, you are becoming less competitive. And you need to rethink the importance of OS architecture.

FLAT ARCHITECTURE: A DEVELOPMENT BOTTLENECK

To start, let's look at the traditional ”flat” architecture still used by many homegrown and commercial embedded OSs. As shown in Figure 1, this architecture folds all software modules into the same address space as the OS kernel; there's no memory protection whatsoever. As a result, any module, no matter how trivial, can overwrite memory used by the kernel - and crash the entire system. All it takes is a single programming error, like an invalid C pointer.

Obviously, this architecture leaves little room for error. Which isn't a problem for a simple embedded design, where most problems can be tracked down during integration testing. But what if your project grows to encompass 50, 100, or 1000 software modules? With thousands of lines of code and hundreds of active execution threads? If a stray pointer crashes your system, where do you start looking? Even the best tools may not identify which module was at fault. And since no one person can understand the entire code base, even experienced kernel programmers can take days, weeks, sometimes months to locate the bug.

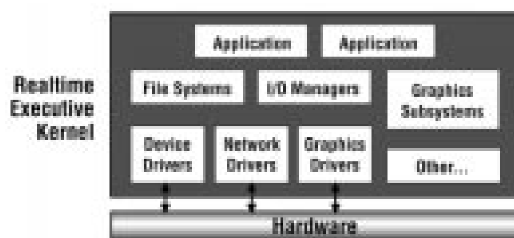

Figure 1. A flat architecture provides no memory protection

Extensive retesting

Now let's say you've fixed the bug, or perhaps added a minor feature. Chances are, you'll have to retest your entire software suite. The reason is simple: Every time you make a code change under a flat architecture, you have to relink the entire runtime image, which creates a new image with different memory-address offsets. As a result, any module that had been overwriting an unused data area may now overwrite a critical data area used by another module or even the OS kernel. This problem can arise from adding just one line of code.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

AD NATIONAL INSTRUMENTS

Reduced prototyping

Flat architecture also limits your ability to try out new design ideas. Think about it: Every time you change code, you have to rebuild the entire runtime image. And every rebuild can, in larger applications, take several hours. As a result, prototyping even a minor feature can be time-consuming.

The performance myth

With a flat architecture, bugs can take so long to locate that your QA team is forced to release unstable code. In fact, as your code base grows, you can end up spending significantly more time on QA than on developing new features or products.

Why, then, do so many OSs continue to use flat architecture? One reason is historical: Until recently, most embedded processors lacked an integrated memory management unit (MMU). Another reason is performance: Many commercial and inhouse OS developers find it difficult to support the additional overhead of the MMU. In fact, some claim that, for the sake of performance, memory protection must be sacrificed.

As it turns out, this isn't necessarily true. As we'll discuss later, not only can a well-designed OS provide extensive MMU protection, it can also deliver performance equal to, or exceeding, that of a conventional, flat architecture OS.

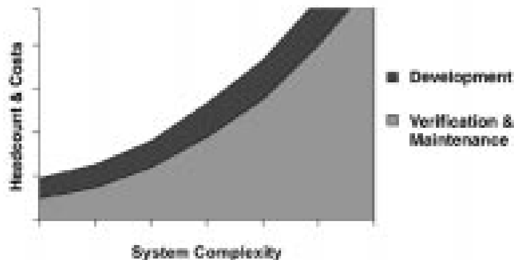

Figure 2. With flat architecture, testing and maintenance increase exponentially as your code base grows

MONOLITHIC ARCHITECTURE: MEETING THE PROBLEM HALFWAY

In an attempt to address the problems of flat architecture, a few embedded OS vendors have adopted a monolithic kernel architecture; see Figure 3. In this architecture, every application module runs in its own memory-protected address space. If an application tries to overwrite memory used by another module, the MMU will trap the fault, allowing the developer to identify where the error occurred.

At first glance, this looks good. The embedded developer no longer has to follow blind alleys, looking for subtle bugs in application code. But there's a catch. All low-level modules - file systems, protocol stacks, drivers, and so on - remain linked to the same address space as the kernel. A single memory violation in just one driver can still crash the system, leaving little or no trace of the error.

That's a problem, since embedded developers spend much of their time developing low-level components. A telecom switch, for example, has a large portion of protocols and drivers for custom hardware. As a result, the problems associated with flat architecture still plague the embedded developer: days or weeks hunting down corrupt C pointers, extensive retesting for every code change, and the potential for even a trivial module to bring down the system.

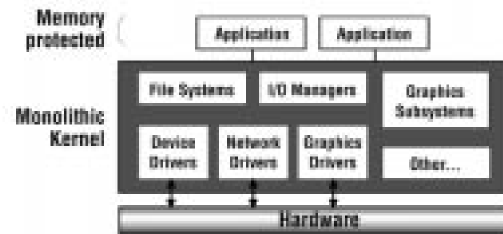

Figure 3. Monolithic architecture provides memory protection, but only for applications

UPM ARCHITECTURE: THE BUG STOPS HERE

To eliminate these development bottlenecks, an OS has to implement Universal Process Model (UPM) architecture.

As shown in Figure 4, a universal process model implements a small set of core services within the kernel itself - such as scheduling, IPC, and initial handling of interrupts. All other system services are provided through optional, add-on processes. As a result, every driver, protocol, file system, I/O manager, and graphics subsystem can run in its own memory-protected address space.

Right away, you can see that this architecture boosts reliability. First, the OS kernel contains very little code that could go wrong. Second, it is highly unlikely that any module - even a poorly written driver running at the highest privilege level - can corrupt the kernel.

Better yet, every module now runs as an independent process, which means you can start, stop, modify, or upgrade any part of your software system at will, without a reboot or a kernel rebuild. Let's see how that streamlines the development cycle - and translates into significantly higher availability at run time.

ADVANTAGES AT DEVELOPMENT TIME

To start, we'll look at how embedded developers spend much of their time - writing device drivers.

With flat or monolithic OS architecture, all driver code is bound to the kernel. As a result:

1. Every time you fix a driver, you typically have to rebuild the kernel and reboot (see Figure 5).
2. Every time you rebuild in a multiuser system, you have to kick off all the other developers. (Since this often isn't possible during normal working hours, driver developers tend to work late at night.)
3. To debug, you have to use low-level kernel tools, instead of easier-to-use source-level tools. (This means that, in addition to losing sleep, driver developers also have to become kernel "gurus.")
4. If a memory corruption occurs, there's no reliable way to pinpoint it.

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

Now compare this to UPM architecture, where a driver runs as a separate, memory-protected process:

1. If you change a driver, you simply recompile it, which can take a matter of seconds. No kernel rebuild required.
2. Since the kernel is never rebuilt, no one gets kicked off. Multiple programmers can share the same target simultaneously and driver developers can get a night's sleep.
3. To debug a driver - or virtually any other traditional kernel module - you can use source-level debugging and profiling tools. Writing a driver or even a custom OS extension becomes as easy as writing a standard application module.
4. If a memory violation occurs, the OS can immediately identify the module responsible - at the exact instruction. Rather than waste weeks tracking down the problem, you can spend minutes solving it.

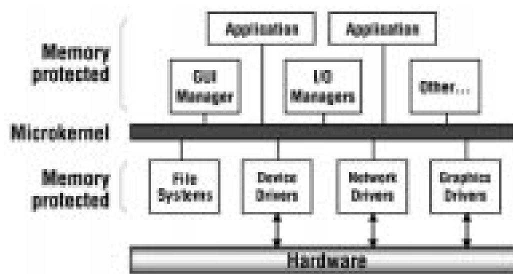

Figure 4. UPM architecture provides memory protection for all software components, including OS modules and drivers

Reusing software and reducing verification

As we've discussed, a single code fix to an embedded system under a conventional OS can result in a different kernel image - and extensive retesting. With UPM, however, the kernel contains only core services, so you can reuse the same kernel binary that has been lab-tested by the OS vendor and field-tested by every other user. Better yet, every module has a linear virtual address space that starts at 0, so you can also reuse the binary image of every unmodified application, driver, OS module, and protocol stack. The result: Most code modifications require that you test only the modules affected - not the entire software suite.

With less time needed for testing, you have more opportunity to add new features or to enhance existing ones, even in the later stages of the design cycle. You can also release different versions of your product more quickly.

Note that some embedded OSs support a limited process model, but don't follow the above approach. Instead of giving each process a virtual address that starts at 0, the OS relies on fixups and offsets to position processes and drivers in memory. As a result, you can't always reuse a binary across products. And if you do, you must be careful that the binary fits into the existing memory-allocation scheme.

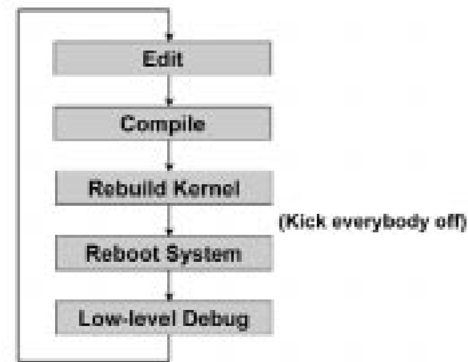

Figure 5. Conventional OS architectures make driver development awkward and time-consuming

Higher return on R&D resources

With conventional OS architectures, every developer may have to learn how each module works in detail, just to avoid trampling on someone else's address space. As the application becomes more complex, programmers can spend more time learning the source tree than enhancing it.

With UPM, programmers don't have to know the system inside out. If a memory corruption occurs, the OS will identify it. Senior developers are freed up to do what they're paid for: solving core problems and adding real value to products. Meanwhile, new developers become productive much sooner. In fact, since UPM allows low-level modules to be debugged with source-level tools, programmers who have written only user applications can now also write drivers, file systems, and so on.

By the same token, UPM makes it easier to outsource software development, since a third-party vendor can contribute code without knowing the source tree intimately. And by providing memory protection between all modules at run time, UPM also makes it less risky to mix the work of outside vendors with code developed inhouse.

Shorter design cycles for large-team projects

Since developers no longer need to know the entire system in detail, UPM makes it easy to divide a large project among smaller, independent teams. This speeds up development in at least two ways. First, small teams are inherently more efficient - 2 or 3 people communicate with each other much faster than 10 or 15 people. A smaller team spends less time on meetings and memos, and more time on the task at hand.

Second, a team can easily develop and test one software subsystem even if related subsystems aren't yet in place. For example, let's say one team is working on module A, another team on module B, and that the two modules will interact closely with each other in the final product. Since every module or group of modules can run as an independent process, the first team can start to write, test, and debug module A without having to wait for the second team to start on module B. All the two teams have to do is first decide on the form of data

the modules will share. The first team can then write a simple "test harness" to test if module A will interact correctly with (the still unwritten) module B.

As mentioned earlier, UPM also allows multiple developers to share the same target system simultaneously. Which means the testing and debugging done by one team won't interfere with - or delay - the testing done by another. Teams work in parallel, rather than in sequence.

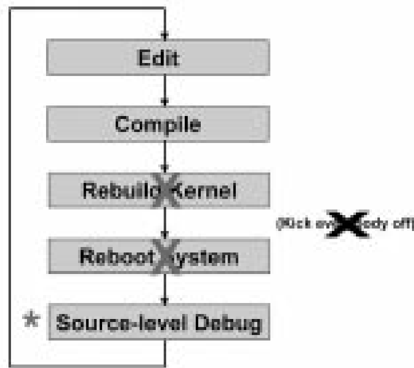

Figure 6. UPM architecture streamlines driver development dramatically

Faster prototyping

With UPM, programmers can also prototype new ideas more quickly. We've seen how arduous prototyping under a flat or monolithic architecture can be, since you have to rebuild the entire runtime image over and over. With UPM, you simply recompile and restart the affected module. As a result, you can often prototype several approaches to a problem in the time it normally takes to prototype just one.

ADVANTAGES AT RUN TIME

UPM virtually eliminates kernel faults. But it has a number of other inherent features that also boost the reliability - and availability - of embedded systems. These include automatic recovery from software errors, hotswapping of both hardware and software, and the ability to distribute components of your application across multiple CPUs.

Increased fault tolerance

No matter how careful you are, some bugs will go undetected - until they show up at run time. With a flat architecture, a reboot is the only way to recover. With a monolithic architecture, you can recover without a reboot, but only if the fault is at the application level. With UPM, however, you can recover without a reboot even if a fault occurs in a driver, protocol stack, or custom OS module. To do this, you use intelligent, user-written mechanisms called software watchdogs.

For example, let's say a driver fails. Instead of forcing a full reset, a software watchdog could:

- simply restart the driver

OR

- restart the driver plus any related processes

In either case, the software designer can determine exactly which processes will be restarted.

Postmortem analysis of software failures

While performing a partial restart or, if necessary, a coordinated system reset, the software watchdog can also collect information about the software failure. For example, if the system has access to mass storage (e.g. flash memory, hard drive, a network link to another computer with a hard drive), the watchdog can generate a process dump file that you can view with source-level debugging tools. This dump file:

- identifies the exact line of code that caused the fault
- allows you to view resources such as variables and a history of function calls

Compare this approach to conventional hardware watchdogs - which simply reset the system without leaving a trace of what went wrong - and the choice is clear. Instead of being at a loss to explain what happened, you can actually work on fixing the problem.

Better still, a software watchdog can monitor system events that may be "invisible" to a hardware watchdog. For example, a hardware watchdog can ensure that a driver is servicing the hardware, but may have a hard time knowing whether other programs are talking to that driver correctly. A software watchdog can cover this hole, and take action before the driver itself shows any problems.

High availability through software hot-swap

Of course, bugs aren't the only thing that bring a system down. Users often have to remove their flat and monolithic systems from service to update drivers, protocol stacks, OS modules, and so on. With UPM architecture, however, you can update these modules "on the fly," without rebooting.

Note that this isn't the same as simply adding new modules to a live system. A few monolithic OSs, for example, will let you dynamically attach drivers to the kernel. But because these drivers then run in kernel space, they can't be removed, restarted, or replaced. In contrast, UPM allows any component to be added or removed at will.

Recovery from hardware faults

With this flexible control over low-level modules, you can also replace outdated or defective hardware, again without rebooting. For example, if an Ethernet card fails, you can terminate the driver, insert a new card, and restart the driver with the new card's parameters. You could even plug in a card that uses another chipset and start the appropriate driver. (Note that the hardware itself must also support hot-swapping.)

Using a similar approach, you can address something as catastrophic as a hard-disk crash. If the local disk fails, you could simply redirect all file operations to a disk on another, network-connected machine. This is a far cry from conventional OS architectures, where even a trivial maintenance task, such as upgrading a mouse, can require a system reset - or a kernel rebuild.

Scalability through distributed processing

One of the best ways to improve reliability is to distribute components of your application across multiple

Copyright 1999 by Real-Time Magazine 99-4 (http://www.realtime-info.com)

CPUs. That way, even a CPU failure won't stop the application from providing service. In fact, as an application grows, you often have no choice but to divide it across CPUs. Not because of reliability (though that may be a factor), but because the application requires more physical interfaces, or simply more processing power, than one CPU can handle.

Unfortunately, conventional OS architectures make this an awkward task, since most or all software modules are bound to the kernel. For example, if you move a protocol stack from one CPU to another, you may have to create, and carefully test, two new kernel images - one for each CPU. And if the OS doesn't provide a transparent means of talking to a module moved to another CPU, then you'll have to recode both the module itself and the modules it communicates with.

As a further complication, it's often difficult to determine which processes should be assigned to which CPU. You may not know until the integration phase that you've chosen to distribute processes in a way that fails to provide optimal performance. At which point it may be too late to recode, rebuild, and retest your software.

UPM sidesteps these problems by decoupling everything from the kernel - every software module is an independent, movable object. And if UPM is implemented so that interprocess communication (IPC) travels transparently across the network, then one process can continue talking to another process even if one of them is moved to another CPU. No code changes or relinking required. In fact, the exact binary of any process can be relocated at any time, even at run time.

This means, of course, that programmers don't have to code with a specific system configuration in mind. No matter how much the final system is scaled up or scaled down, programmers can write their programs just one way. It doesn't matter, for example, whether a hard disk and its associated driver will eventually be located on the local machine or on a remote, network-connected machine. Either way, any process (provided it has the appropriate authority) will be able to access the hard disk transparently, without special code.

BUT WHAT ABOUT PERFORMANCE?

We've seen the benefits of UPM during development time and at run time. Nevertheless, one question remains: Does placing each application, driver, and OS module in its own MMU segment impede performance? The answer is no, not if UPM is implemented correctly.

For example, a UPM operating system like QNX can perform a context switch - the time it takes to stop running one process and to start running another - in just 1.95 msec on a 133MHz Pentium. That's more than enough for virtually any performance-critical application.

RECLAIMING INNOVATION

To return to our original question, your choice of OS does make a difference. Not just to system reliability or performance, but to your very ability to create new products under tight deadlines. Of course, software development tools can also make a difference. But as we've seen, tools can't compensate for the significant overhead of debugging, testing, and maintenance imposed by conventional OS architectures.

By eliminating much of that overhead, UPM architecture encourages a "culture" of innovation. Design teams have the breathing room to add or enhance features, or to create whole new products. Talented team members who like to do something new are encouraged to stay, rather than join another company. And teams themselves can be smaller, more focused, less prone to communication bottlenecks - and, as a result, more responsive to deadlines. Better yet, UPM makes it easier to integrate code developed by third parties, thus providing one more way to handle shrinking development cycles.

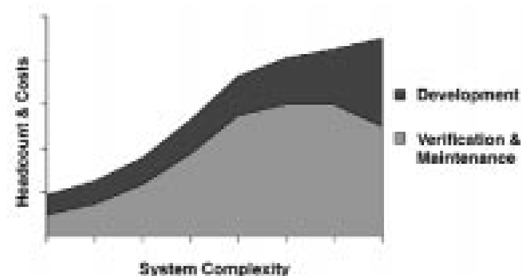

Figure 7. With UPM, you can "flatten the curve" to achieve the next level of competitiveness

Innovation aside, UPM provides the one other thing customers look for: reliability. Unlike systems based on conventional OS architectures, a UPM-based system can recover from software faults and hardware failures, without rebooting. What's more, it lets you update drivers, applications, protocols, even parts of the OS itself, without removing the system from service.

Innovation, time-to-market, reliability. Can an OS help you with all three? The answer is a definite yes - provided, of course, the OS has the right architecture ■

Paul Leroux is currently a technology analyst at QNX Software Systems, where he has served in various roles since 1990. With interests ranging from digital convergence to software design for datacommunications equipment, Paul writes on a variety of topics, including network computing, driver information systems, and embedded web technology.
When not in front of his word processor, Paul can be found swimming, roller blading, and searching for a house large enough to accommodate his ever-growing CD collection.