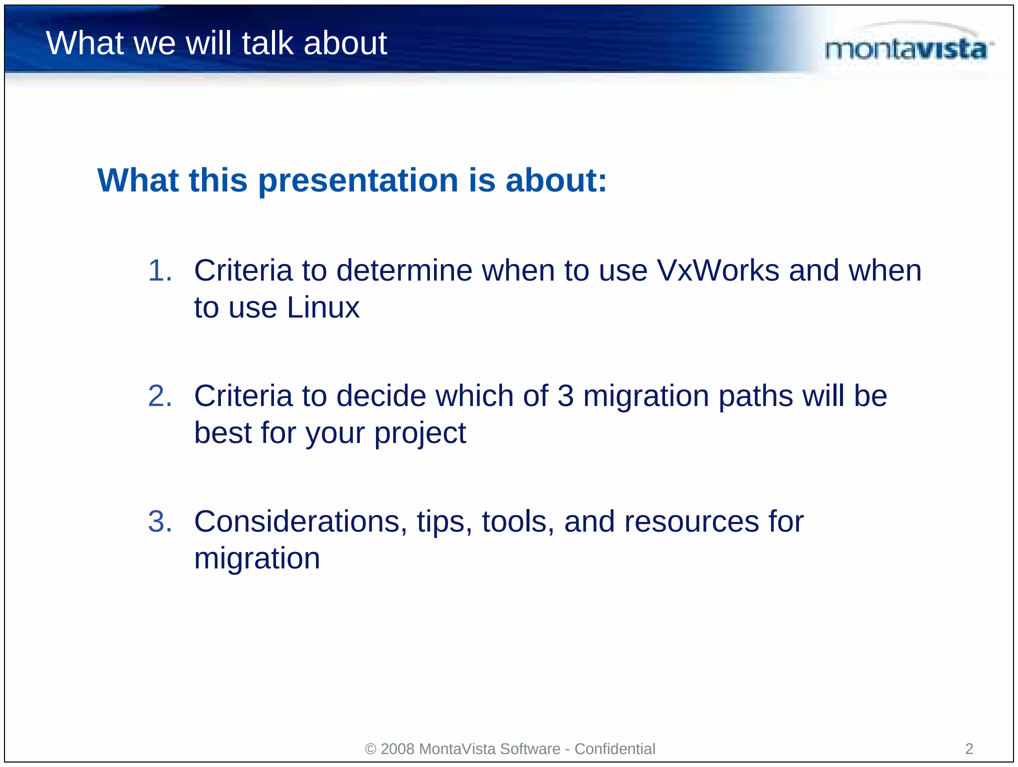
Migrating VxWorks applications to Linux

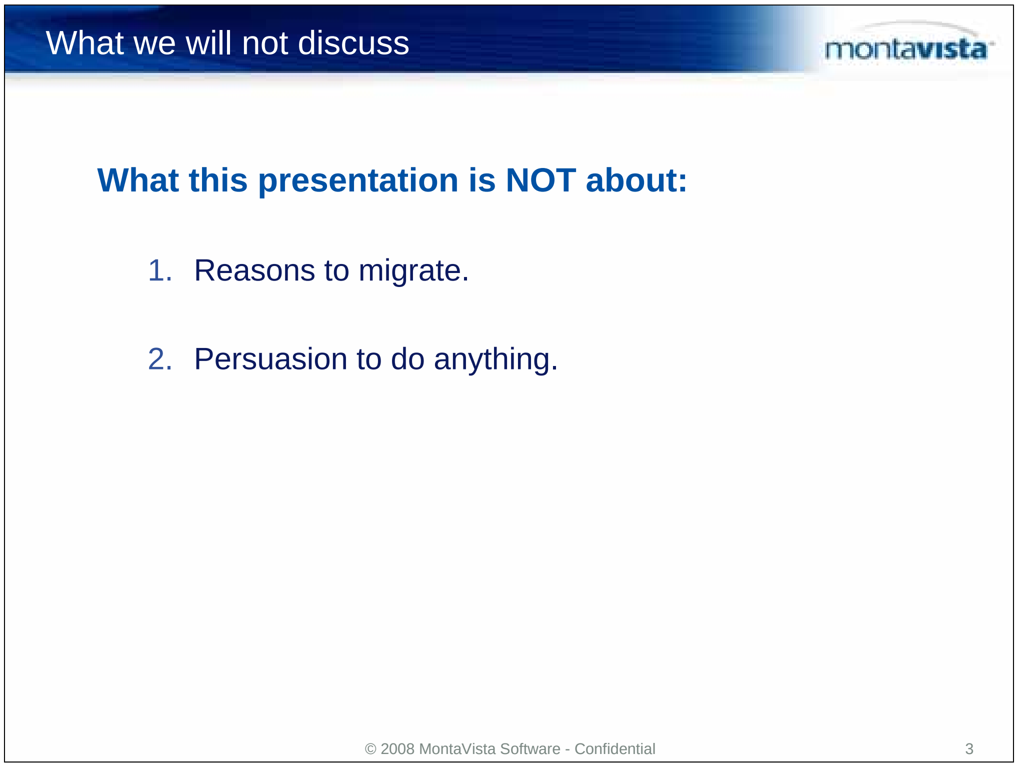
Jim Ready
CTO and founder
MontaVista Software



What we will talk about

What this presentation is about:

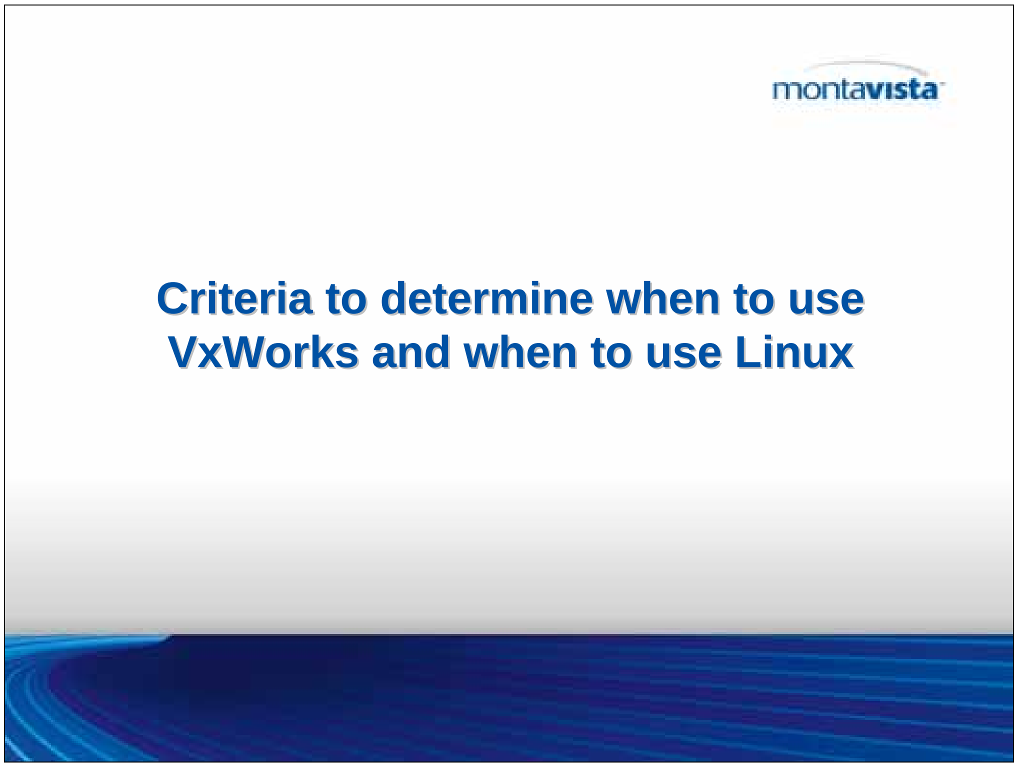
1. Criteria to determine when to use VxWorks and when to use Linux
2. Criteria to decide which of 3 migration paths will be best for your project
3. Considerations, tips, tools, and resources for migration



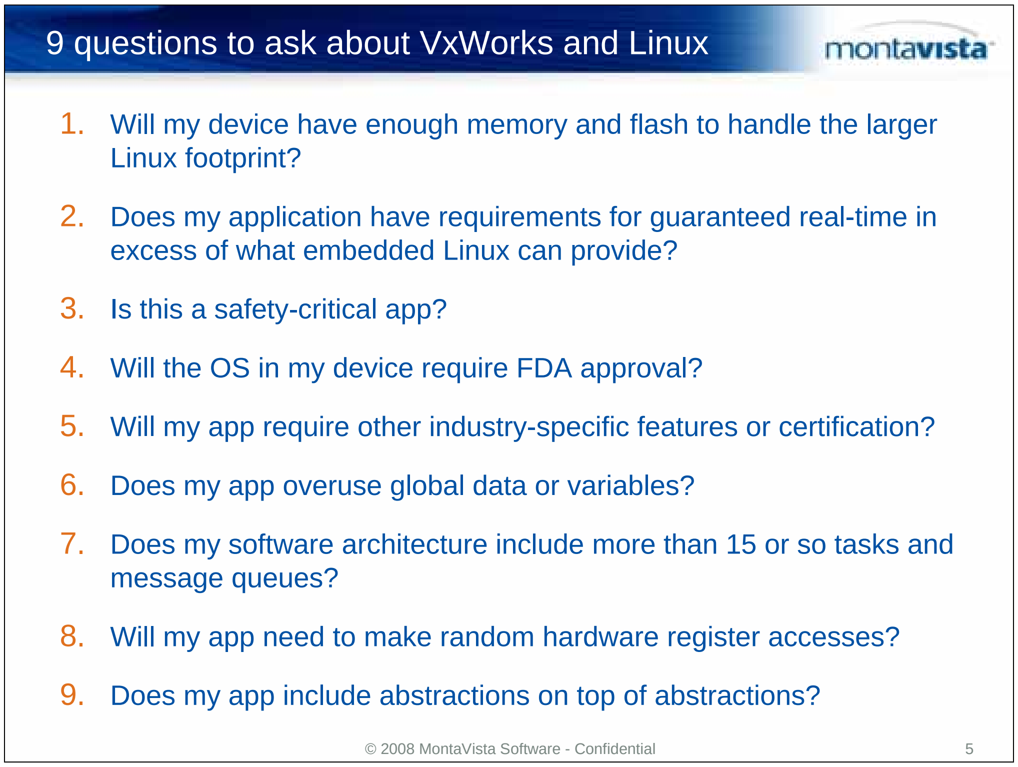
What we will not discuss

What this presentation is NOT about:

1. Reasons to migrate.
2. Persuasion to do anything.



Criteria to determine when to use VxWorks and when to use Linux



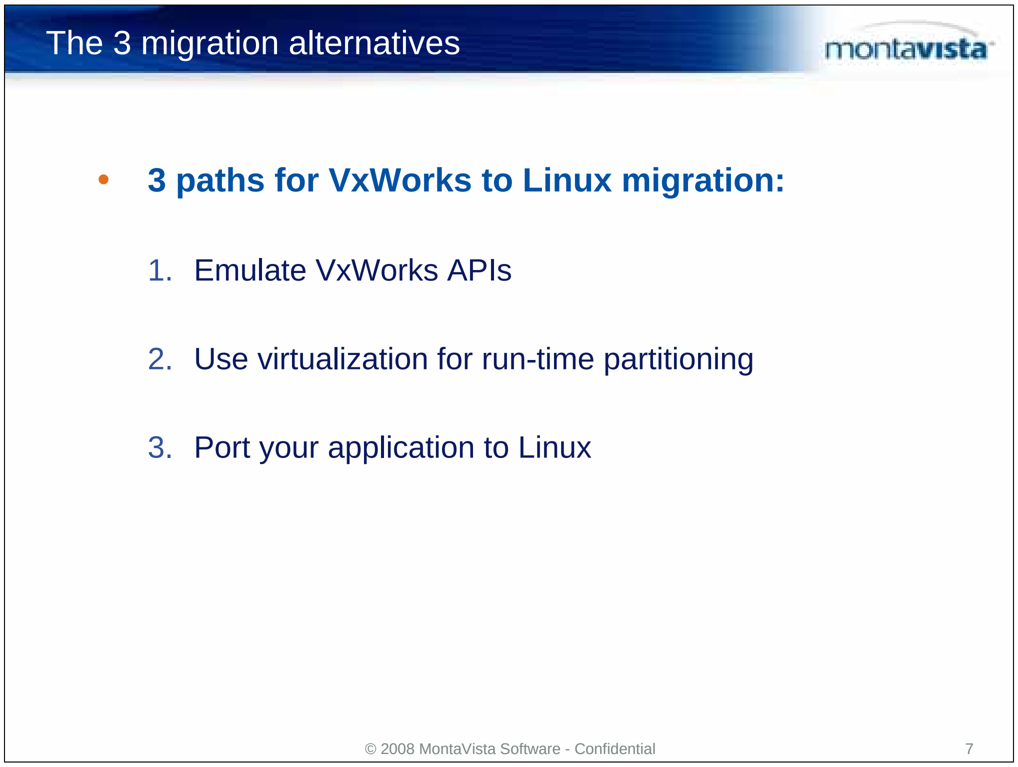
9 questions to ask about VxWorks and Linux

1. Will my device have enough memory and flash to handle the larger Linux footprint?
2. Does my application have requirements for guaranteed real-time in excess of what embedded Linux can provide?
3. Is this a safety-critical app?
4. Will the OS in my device require FDA approval?
5. Will my app require other industry-specific features or certification?
6. Does my app overuse global data or variables?
7. Does my software architecture include more than 15 or so tasks and message queues?
8. Will my app need to make random hardware register accesses?
9. Does my app include abstractions on top of abstractions?

Don’t jump into something you don’t need to have.

1. The VxWorks kernel can be as small as 64kB and can run with 128kB flash. Embedded Linux = less compact.
2. Embedded Linux can perform with sub-50 microsecond response time. If guaranteed hard real-time response is an absolute requirement for your project, you might not want to make your first embedded Linux project your guinea pig. Do a pilot project first.
3. Is your project a DO 178 app? If you have specific safety-critical specifications you must meet, embedded Linux might not be your best bet.
4. Many medical devices use embedded Linux, but if your project is a medical device that requires your software to meet FDA approval, your project may go through the approval process faster with VxWorks.
5. VxWorks may have some certification or industry-specific features that your project requires. If you are looking to switch to embedded Linux, make sure that you can still meet your customers’ needs for certification or industry-specific features. Of course, embedded Linux has certifications and industry-specific features as well.
6. Beware if your app has global data or variables, such as variables shared across multiple tasks. Search for “extern” references in your code. If you find it 5 or more times, it is a warning sign.
7. If your app has 15-20 tasks and message queues, you have a software architecture problem from the Linux point of view. Task switching is more expensive in Linux.
8. Linux does not allow random access to hardware registers.

Which of 3 migration paths will be best for your project?



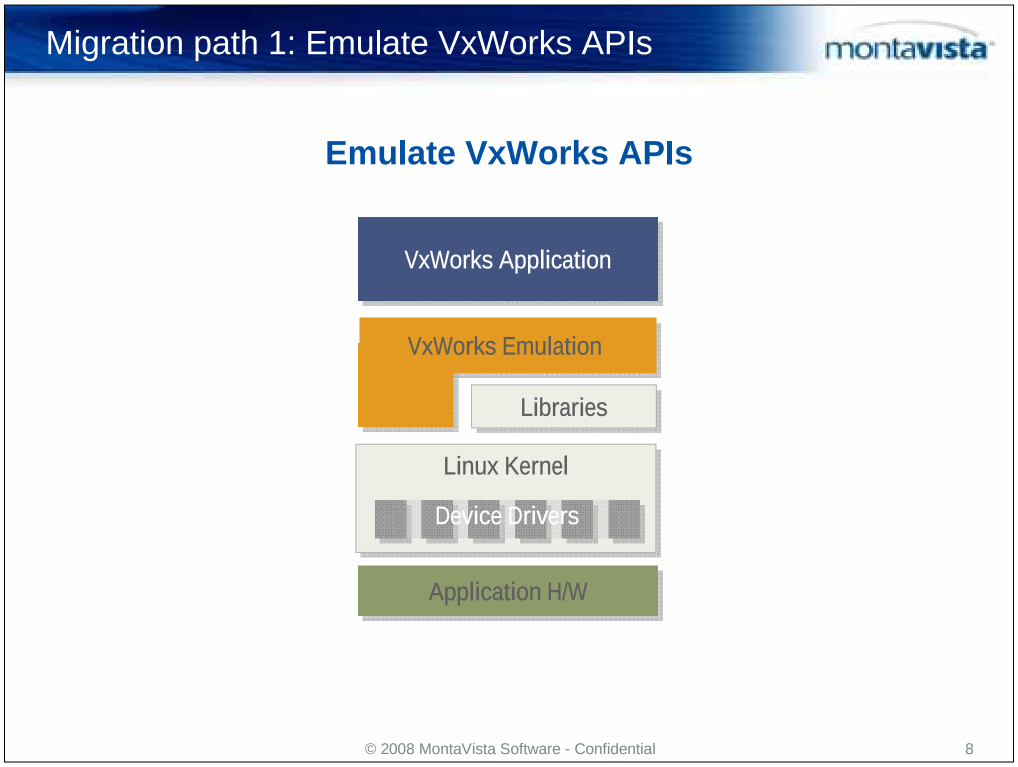
The 3 migration alternatives

- **3 paths for VxWorks to Linux migration:**
  1. Emulate VxWorks APIs
  2. Use virtualization for run-time partitioning
  3. Port your application to Linux

Linux may increasingly find itself in the place of VxWorks, but the architecture of the Linux OS is very different from VxWorks.

Let’s compare and contrast three migration and re-hosting paradigms for legacy software under Linux:

1. RTOS API Emulation over Linux
2. Run-time partitioning with virtualization
3. Full native Linux application port



Migration path 1: Emulate VxWorks APIs

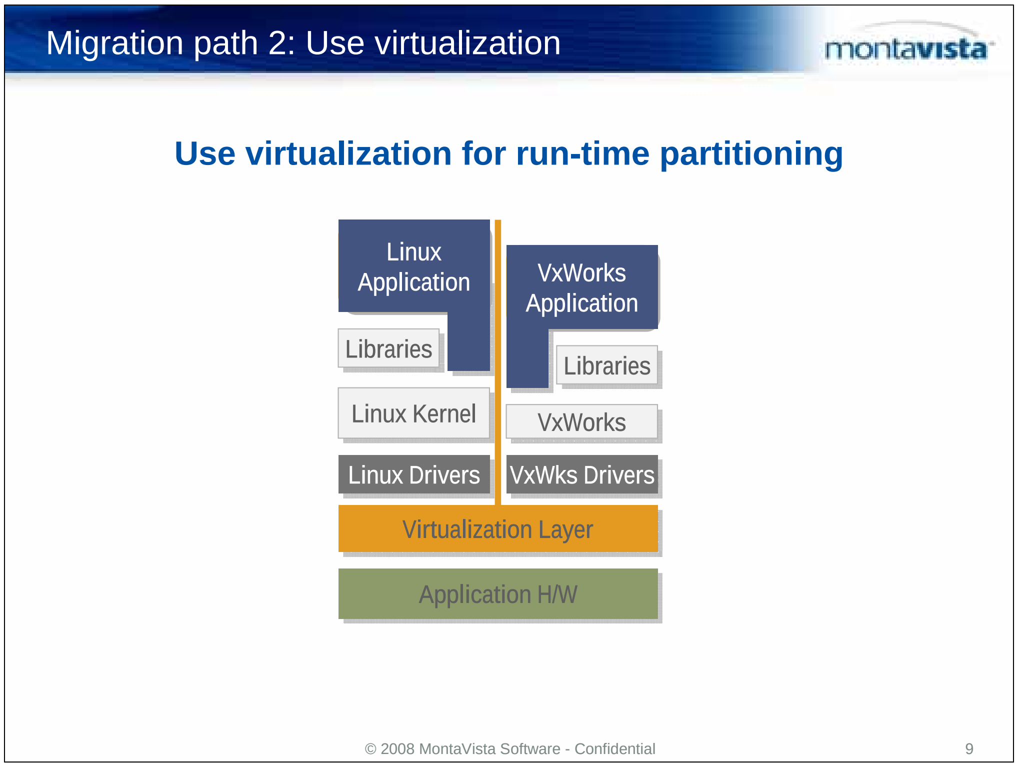
Emulate VxWorks APIs

8

For legacy VxWorks applications to host and execute on Linux, a Linux-hosted run-time must exist to service RTOS system calls and other APIs. Many RTOS entry points and stand-alone compiler library routines have exact analogs in Linux and in the glibc run-time library, but not all do. In many cases new code must intervene to emulate missing functionality. Even when analogous APIs exist, they can present parameters that differ in type and number.

Legacy RTOSes can implement hundreds of system calls and library APIs; VxWorks documentation describes over one thousand unique functions and subroutines. Real applications use only a few dozen unique RTOS APIs and call functions from standard C/C++ libraries for the rest of their operations.

To emulate these interfaces for purposes of migration, developers only need a core subset of RTOS calls. Many OEMs choose to build and maintain emulation lightweight libraries themselves; others look to more comprehensive commercial offerings from vendors like MapuSoft. There also exists an open source project called v2lin that emulates several dozen commonly used VxWorks APIs. v2lin originated at MontaVista Software and is now maintained by others. The project was recently rearchitected to work with newer POSIX-compliant versions of glibc.



Migration path 2: Use virtualization

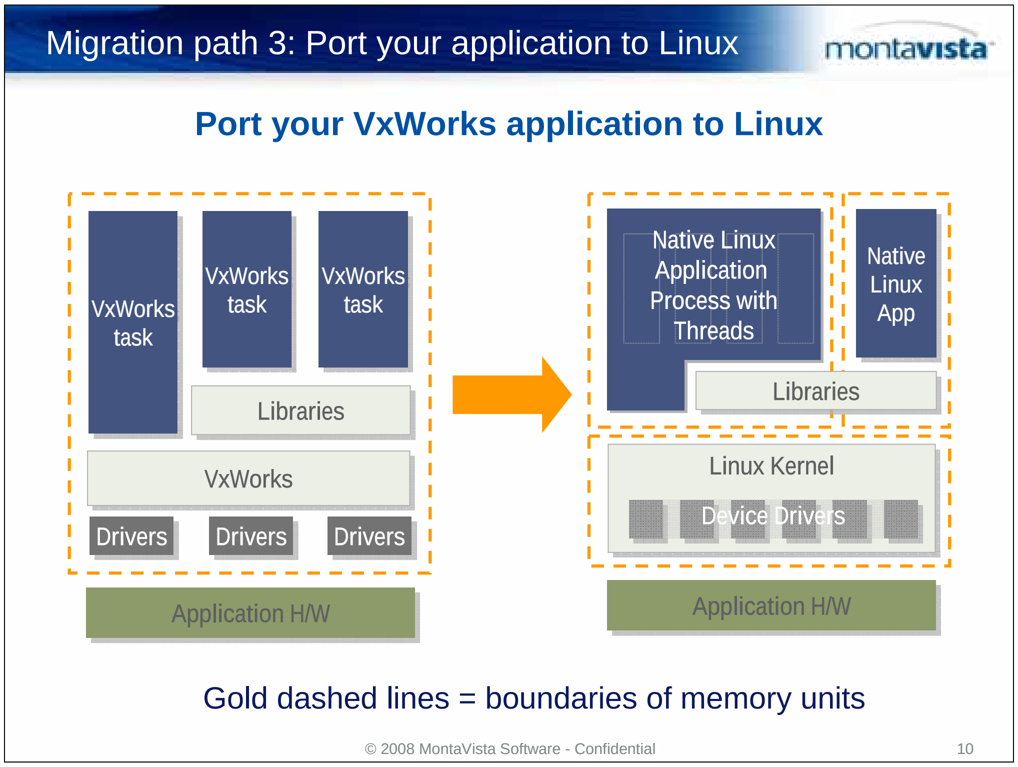
Use virtualization for run-time partitioning

Virtualization involves the hosting of one operating system or running as an application “over” another virtual platform, where a piece of system software (running on “bare metal” or as a hosted application) enables the execution of one or more “guest” OS instances. In enterprise computing, Linux-based virtualization technology is a mainstream feature of data centers, but virtualization also finds many applications in embedded systems.

Embedded virtualization entails partitioning of CPU, memory and other resources to host VxWorks or another RTOS and one or more guest “application” OSes, usually Linux, to run higher-level software. Virtualization can support migration by allowing an VxWorks application and VxWorks itself to run mostly intact in a new design, with Linux executing in its own partition. This arrangement (see the diagram) is useful when legacy code has dependencies on VxWorks APIs and on performance characteristics, e.g., real-time performance or specific implementations of protocol stacks. Embedded virtualization as such represents a short and solid bridge from legacy code to new Linux-based designs, but that bridge exacts a toll: OEMs will continue to pay legacy VxWorks run-time royalties and will also need to negotiate a commercial license from the VM supplier.

A wide range of options exists for virtualization, including the mainstream KVM (Kernel-based Virtualization Manager) and Xen. Embedded-specific paravirtualization solutions are available from a range of independent software vendors, including Open Kernel Labs, Trango, and VirtualLogix. You might also look at the open source projects Adeo and Xenomai.

Processor architectures can jeopardize this plan; be careful.



Migration path 3: Port your application to Linux

Port your VxWorks application to Linux

Gold dashed lines = boundaries of memory units

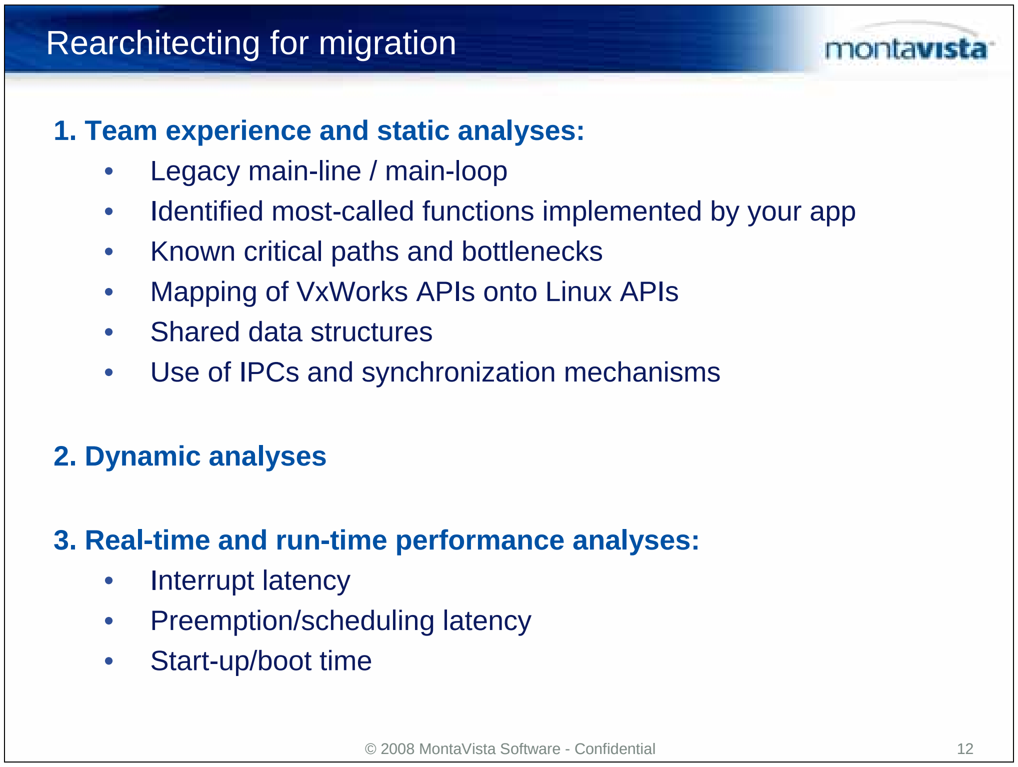
10

Emulation and virtualization can provide straightforward migration paths for prototyping, development and even deployment of legacy VxWorks applications running on Linux. They have the drawback, however, of including additional code, infrastructure and licensing costs. Instead, “going native” on Linux reduces complexity, simplifies licensing, enhances portability and performance.

Your choice need not be exclusive. The first time OEMs approach migration they are likely to leverage emulation and virtualization technologies. As they learn and acquire greater familiarity with development tools and run-time attributes of Linux, OEMs can reengineer legacy applications incrementally for native Linux execution.

One approach is to choose individual legacy programs for native migration and to host them under Linux in separate processes. This technique works best with software exhibiting minimal or formalized dependencies on other subsystems. Another sensible practice is to implement new functionality only as native code, even if employing emulation or virtualization.

Considerations, tips, tools, and resources for migration



Rearchitecting for migration

1. Team experience and static analyses:
- Legacy main-line / main-loop
- Identified most-called functions implemented by your app
- Known critical paths and bottlenecks
- Mapping of VxWorks APIs onto Linux APIs
- Shared data structures
- Use of IPCs and synchronization mechanisms

2. Dynamic analyses

3. Real-time and run-time performance analyses:
- Interrupt latency
- Preemption/scheduling latency
- Start-up/boot time

© 2008 MontaVista Software - Confidential 12

Rearchitecting – Where to Begin

Optimizing application and system code for a new platform can be a daunting task. You should consider three approaches or focus areas:

1) Your team experience and static analyses

Your organization probably already employs some form of static analysis tools and disciplines. Your team also possesses a wealth of knowledge from real-world experience with the code undergoing migration. Using this mix of tools and talent, begin by reviewing:

- Legacy main-line / main-loop
- Identified most-called functions implemented by your application (top 15%)
- Known critical paths and bottlenecks

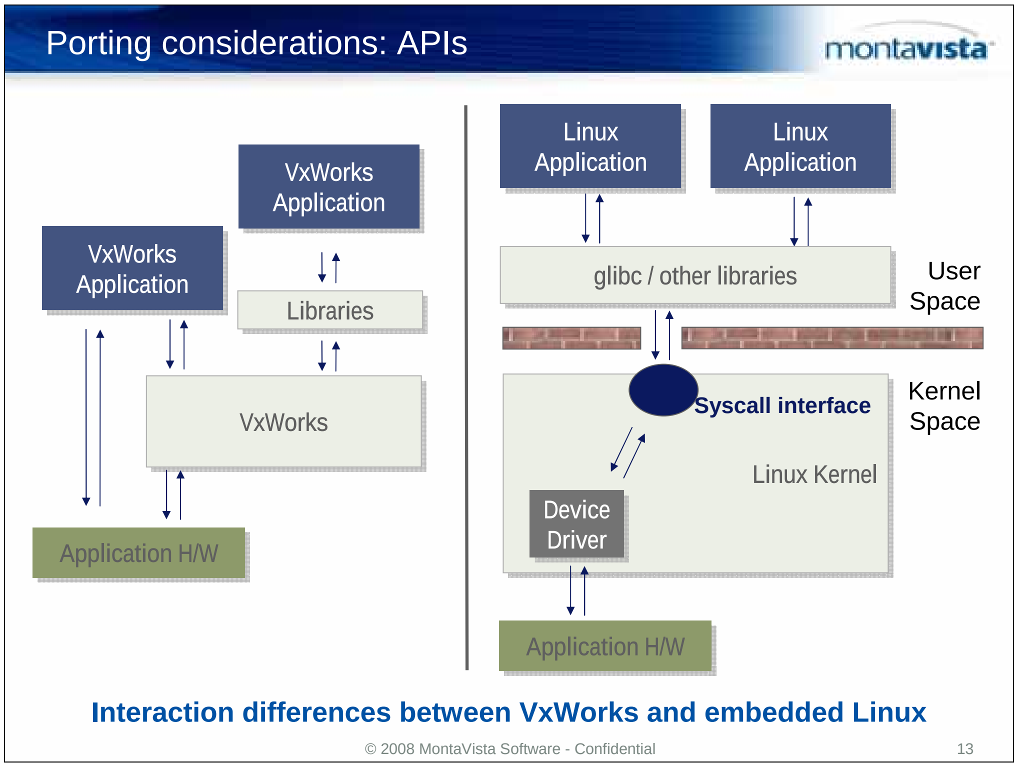
and by examining:

- Mapping of VxWorks APIs onto Linux APIs
- Shared data structures
- Use of IPCs and synchronization mechanisms

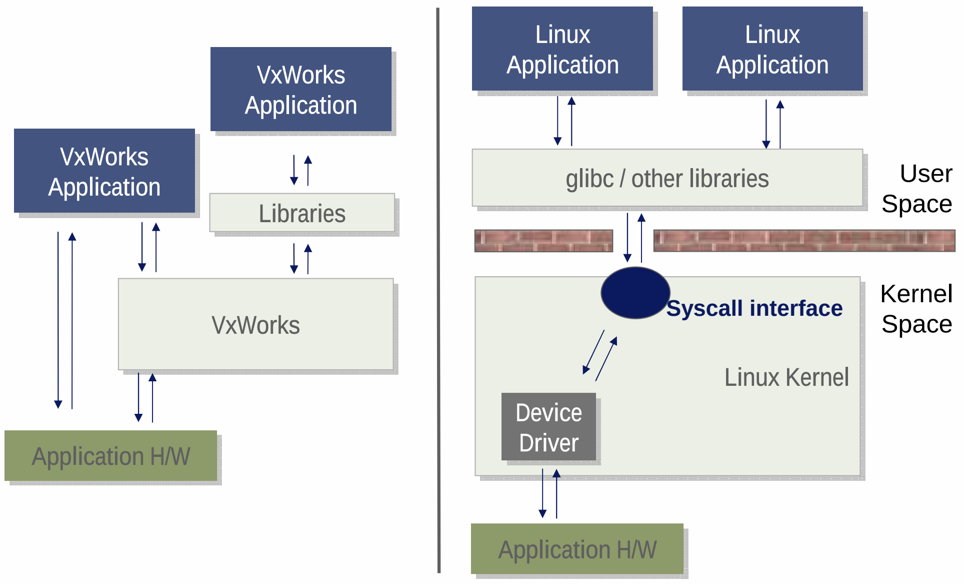
This level of analysis is the best way to highlight primary candidates for re-architecting.

2) Dynamic analyses

Using dynamic analysis will confirm raw static frequency analysis and provide guidance on where to spend your engineering budget in optimizing and re-architecting. A key exercise is to compare where your legacy application spent



Porting considerations: APIs

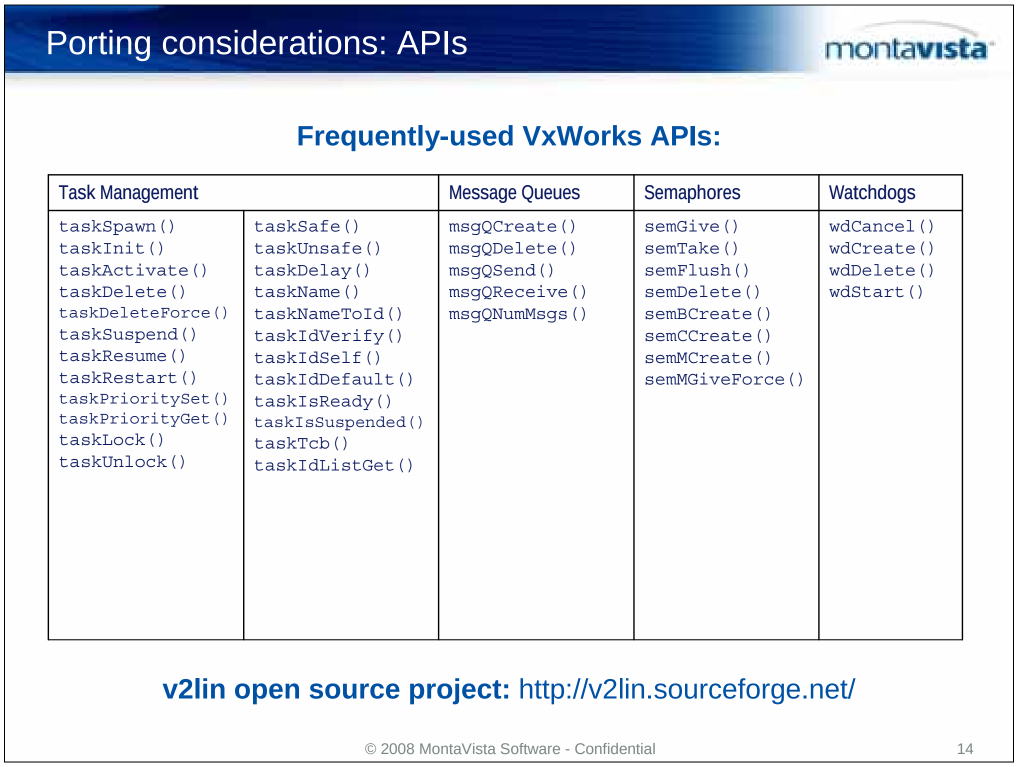


Interaction differences between VxWorks and embedded Linux

The good news is that Linux features perhaps the richest array of APIs of any embedded operating system. The bad news is that your code may exploit VxWorks calls and features that do not readily translate into the Linux model.

For performance reasons, VxWorks system calls were implemented by direct subroutine calls instead of the traps or exceptions used in “real” protected OSes like Linux. In practice, application code accesses to system calls occur through libraries that either act as “wrappers” for the system calls or even implement entire functions inside the library code without ever calling the kernel. Examples of the first case are task creation and scheduling calls, like VxWorks taskInit(); examples of the second include library-based threading schemes on older UNIX systems or so-called “green threads” in Java.

Your VxWorks app probably makes no distinction between direct system calls and library functions and may leverage dozens or even hundreds of available APIs under VxWorks. Kernels like VxWorks, (and pSOS, VRTX, and Nucleus) have accrued hundreds, even thousands of APIs in their decades of commercial existence and it is not practical to address the mass of those APIs. A more pragmatic approach is to translate and emulate a clean core set of the four or five dozen most common calls, and to leave the rest for *ad hoc* translation and implementation.



Porting considerations: APIs

Frequently-used VxWorks APIs:

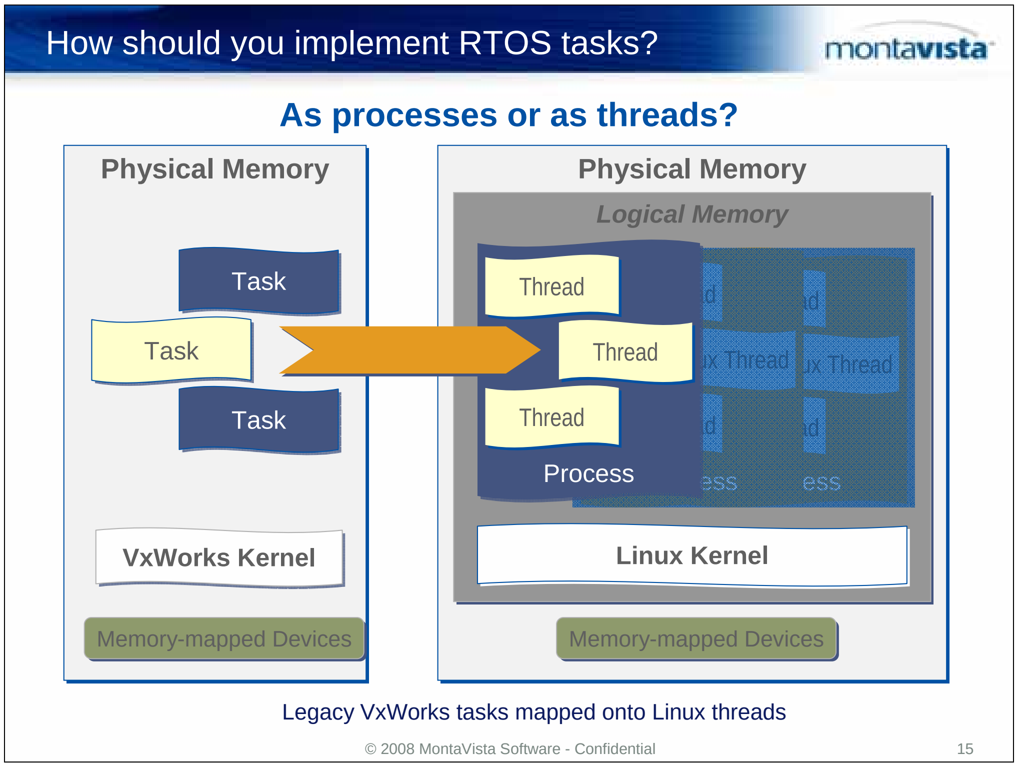
Task Management		Message Queues	Semaphores	Watchdogs
taskSpawn()	taskSafe()	msgQCreate()	semGive()	wdCancel()
taskInit()	taskUnsafe()	msgQDelete()	semTake()	wdCreate()
taskActivate()	taskDelay()	msgQSend()	semFlush()	wdDelete()
taskDelete()	taskName()	msgQReceive()	semDelete()	wdStart()
taskDeleteForce()	taskNameToId()	msgQNumMsgs()	semBCreate()	
taskSuspend()	taskIdVerify()		semCCreate()	
taskResume()	taskIdSelf()		semMCreate()	
taskRestart()	taskIdDefault()		semMGiveForce()	
taskPrioritySet()	taskIsReady()			
taskPriorityGet()	taskIsSuspended()			
taskLock()	taskTcb()			
taskUnlock()	taskIdListGet()			

v2lin open source project: http://v2lin.sourceforge.net/

VxWorks includes standard call sets from POSIX or BSD, but those APIs exist primarily as “window dressing”. Most developers use, and Wind River recommends, its proprietary, closed VxWorks APIs, and it is those proprietary calls that lock projects into long-term commitments to that platform.

Many of the APIs in embedded designs are simply standard C language libraries that either directly implement functionality or act as wrappers for system calls. On Linux, you’ll find libc/glibc completely familiar, although more comprehensive than the VxWorks library implementation.

In many cases, VxWorks offer calls completely or nearly identical to Linux APIs. Because the original VxWorks developers were also UNIX programmers, VxWorks features entry points like open, write, etc. Such calls will either map one-to-one completely unchanged, will be hidden by compiler library wrappers, or may require some minimal tweaking with #defines in header files.



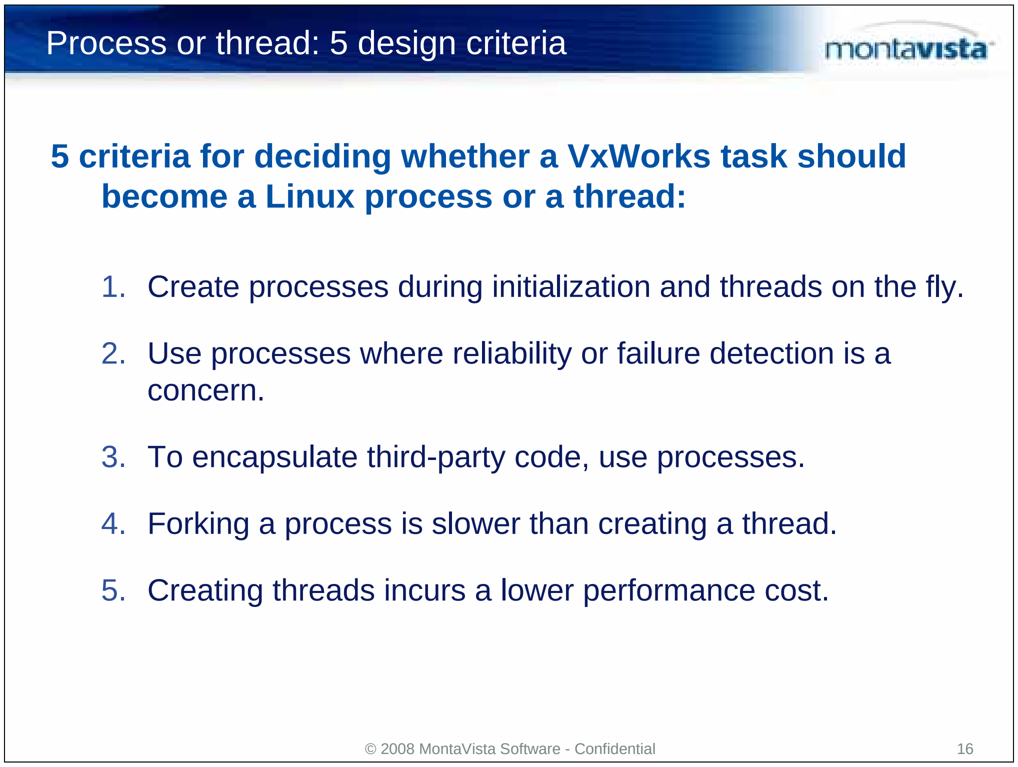
How should you implement RTOS tasks?

As processes or as threads?

Legacy VxWorks tasks mapped onto Linux threads

The previous architecture descriptions suggest a straightforward architecture for porting VxWorks code to Linux: the entirety of VxWorks application code (minus kernel and libraries) migrates into a single Linux process; VxWorks tasks translate to Linux threads; the physical memory space maps into Linux virtual address spaces – a multi-board or multiple processor architecture (like a VME rack) migrates into a multi-process Linux application, as in the illustration here.

Whether you use RTOS emulation kits for VxWorks or for other RTOSes, or perform the port unaided, you will ultimately have to make decisions regarding whether to implement RTOS tasks as processes or as threads. While the Linux kernel treats both processes and threads as co-equal for scheduling purposes, there are different APIs for creating and managing each type of entity, and performance and resource costs (and benefits) associated with each.



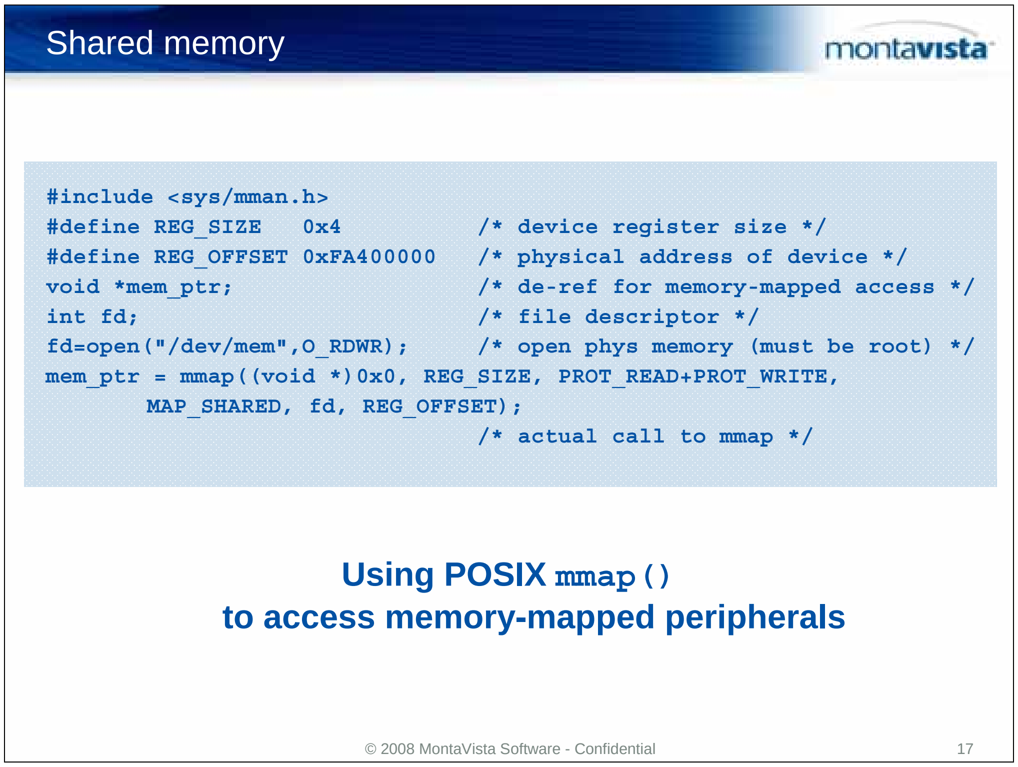
Process or thread: 5 design criteria

5 criteria for deciding whether a VxWorks task should become a Linux process or a thread:

1. Create processes during initialization and threads on the fly.
2. Use processes where reliability or failure detection is a concern.
3. To encapsulate third-party code, use processes.
4. Forking a process is slower than creating a thread.
5. Creating threads incurs a lower performance cost.

While a simple port will typically map VxWorks tasks onto Linux threads, subsequent modifications will require decisions on the part of the developer. Following are some heuristics for making this decision:

1. In general, create processes during initialization and threads on the fly.
2. Use processes for greater reliability and where health monitoring and failure detection (via **SIGCHLD**) is a concern.
3. Employ processes to encapsulate third-party code; if that code blows up, it can do much less harm and can always be restarted.
4. Calls to fork() a process can run into tens of milliseconds; thread creation is much faster and executes in tens of microseconds. This is very dependent on your architecture.
5. Creating entirely new processes / loading new programs (via calls like execv()) carries the heaviest cost, since it accesses file systems to load an executable image and must create a new virtual address space.


Shared memory

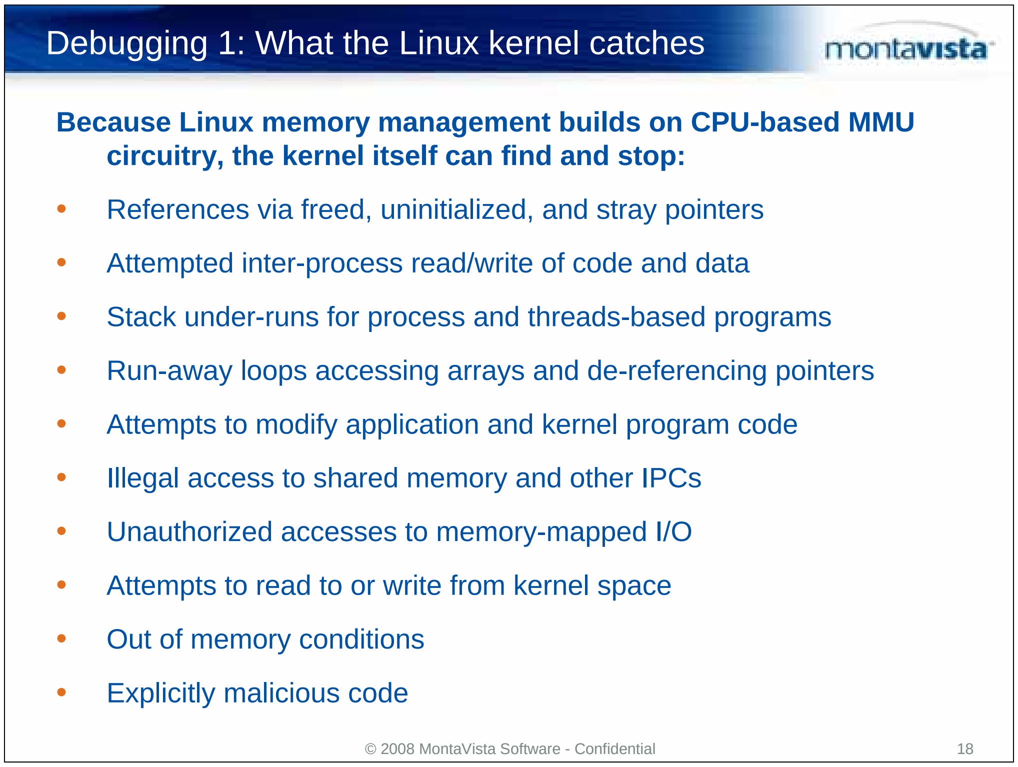
```
#include <sys/mman.h>
#define REG_SIZE   0x4           /* device register size */
#define REG_OFFSET 0xFA400000    /* physical address of device */
void *mem_ptr;                   /* de-ref for memory-mapped access */
int fd;                          /* file descriptor */
fd=open("/dev/mem",O_RDWR);      /* open phys memory (must be root) */
mem_ptr = mmap((void *)0x0, REG_SIZE, PROT_READ+PROT_WRITE,
       MAP_SHARED, fd, REG_OFFSET);
                                 /* actual call to mmap */
```

Using POSIX `mmap()`
to access memory-mapped peripherals

Most VxWorks programs make use of informally shared data structures. Some RTOSes offer a thin formalism for sharing blocks of memory, but without MMU-based memory protection, such schemes are artificial at best and unneeded overhead at worst.

Linux, with its POSIX process model and strictly enforced virtual addressing, offers applications a robust shared memory capability. Processes can share memory with each other and with drivers via calls like the POSIX.1b mmap() and the SVR4 shmget() interfaces, and govern the usage with the synchronization and mutual exclusion mechanisms described above.

To deal with shared memory among boards in a multiprocessor system, such as a VME cage, VxWorks uses shared memory mechanisms (with VxMP). Embedded Linux-based apps again leverage the mmap() API, allowing for shared memory across VME, CompactPCI, ATCA and other interconnects to be mapped into Linux virtual address spaces.


Debugging 1: What the Linux kernel catches

Because Linux memory management builds on CPU-based MMU circuitry, the kernel itself can find and stop:

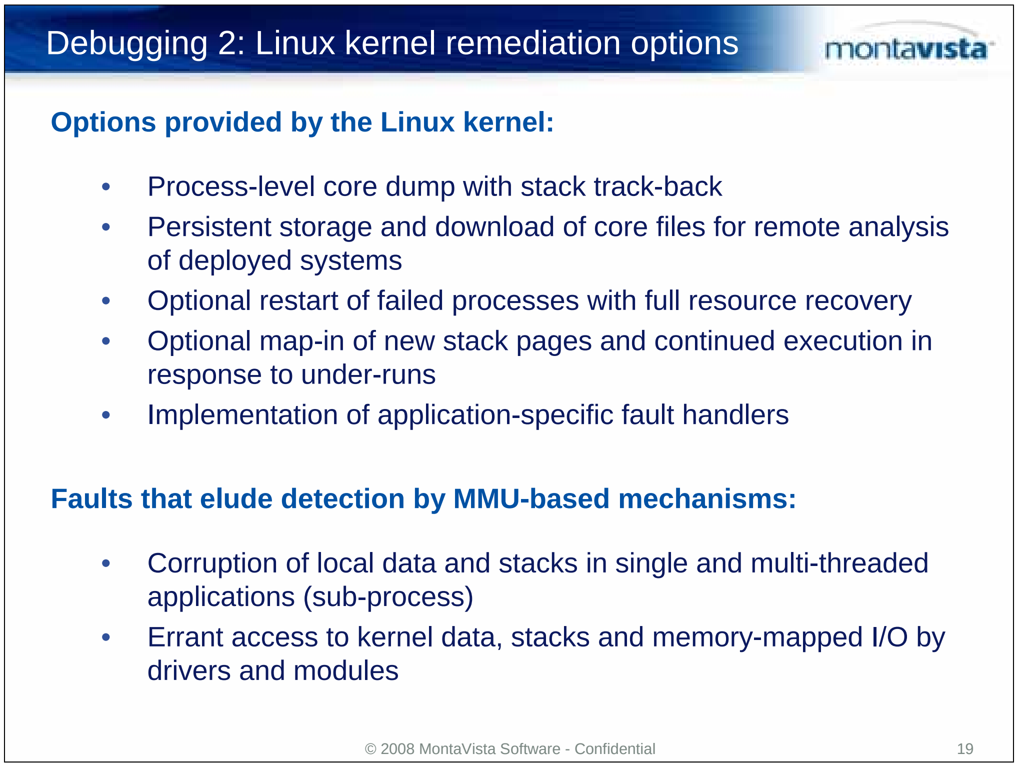
- References via freed, uninitialized, and stray pointers
- Attempted inter-process read/write of code and data
- Stack under-runs for process and threads-based programs
- Run-away loops accessing arrays and de-referencing pointers
- Attempts to modify application and kernel program code
- Illegal access to shared memory and other IPCs
- Unauthorized accesses to memory-mapped I/O
- Attempts to read to or write from kernel space
- Out of memory conditions
- Explicitly malicious code

18

Another important aspect of migration is how it impacts the development cycle, in particular, how it changes debugging. The short answer is that cross platform debugging of application code doesn't change that much. The longer answer is that just as Linux has both evolutionary and revolution impact on embedded code deployment, so does it also alter the processes and challenges to debugging that code.

You should know that the Linux run-time environment in user space includes one of the best debugging tools in existence: the Linux kernel itself. The kernel itself is a good debugger for apps.

Except for minimal configuration for caching, a legacy RTOS like VxWorks mostly ignores on-chip memory management unit (MMU) capabilities. Linux memory management builds on CPU-based MMU circuitry.



Debugging 2: Linux kernel remediation options

Options provided by the Linux kernel:

- Process-level core dump with stack track-back
- Persistent storage and download of core files for remote analysis of deployed systems
- Optional restart of failed processes with full resource recovery
- Optional map-in of new stack pages and continued execution in response to under-runs
- Implementation of application-specific fault handlers

Faults that elude detection by MMU-based mechanisms:

- Corruption of local data and stacks in single and multi-threaded applications (sub-process)
- Errant access to kernel data, stacks and memory-mapped I/O by drivers and modules

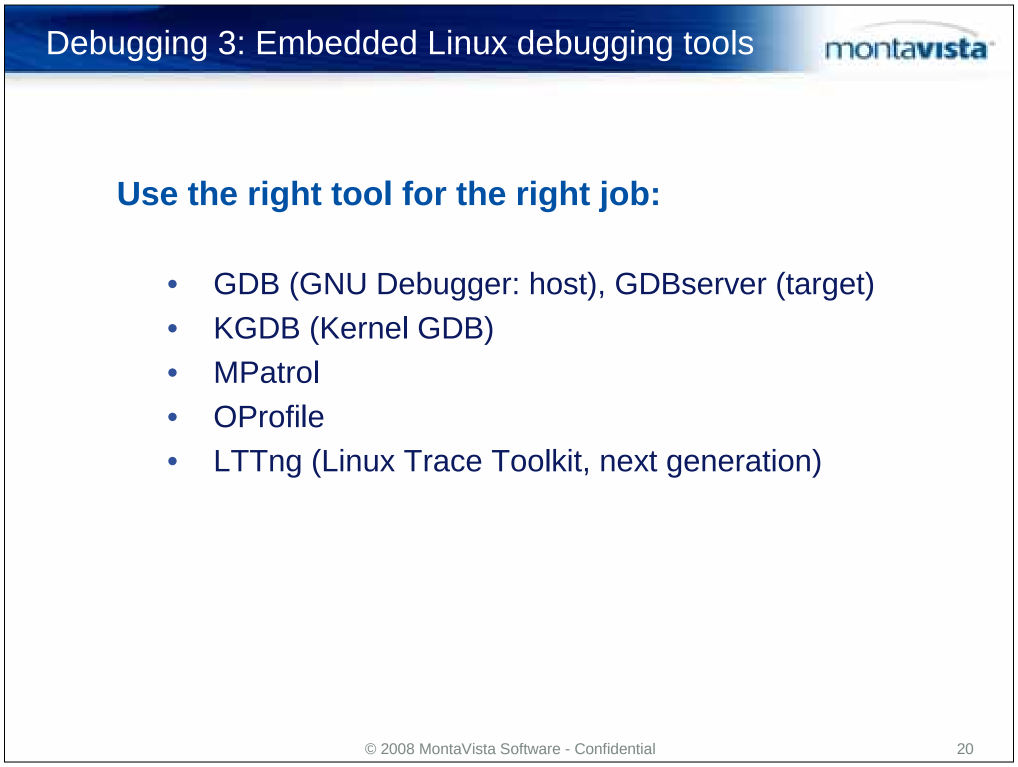
Not only does the Linux kernel catch most of these fault conditions, it provides a range of remediation options, both on the test bench and in the field:

- Process-level core dump with stack track-back (use GDB to find offending source code; CGE even has a kernel core dump.)
- Persistent storage and download of core files for remote analysis of deployed systems
- Optional restart of failed processes with full resource recovery (memory, file handles, etc., including mutexes if real-time)
- Optional map-in of new stack pages and continued execution in response to under-runs
- Implementation of application-specific fault handlers

The Linux kernel, while a powerful debugging and hardening tool, is not a “magic bullet”. There are still several types of fault conditions that elude detection and remedy by MMU-based process-centric mechanisms:

- Corruption of local data **and stacks** in single and multi-threaded applications (sub-process)
- Errant access to kernel data, stacks and memory-mapped I/O by drivers and modules

Finding and remedying these types of bugs is best accomplished with debugging tools.



Debugging 3: Embedded Linux debugging tools

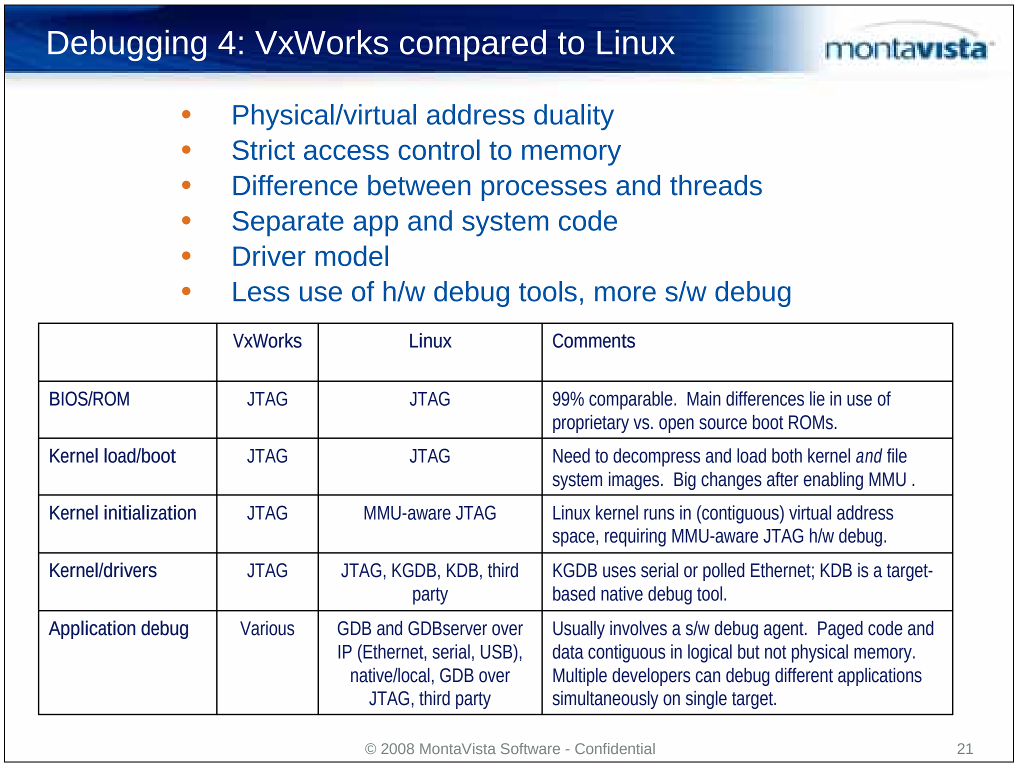
Use the right tool for the right job:

- GDB (GNU Debugger: host), GDBserver (target)
- KGDB (Kernel GDB)
- MPatrol
- OProfile
- LTTng (Linux Trace Toolkit, next generation)

© 2008 MontaVista Software - Confidential 20

Many VxWorks developers liked the “all in one” concept behind Tornado. However, the downside of a single-vendor multi-function environment was that developers had limited, license-enforced choices for the tools they could use and the workflows they could follow.

By contrast, Linux offers developers a wide range of tools and techniques for system bring up and debugging of platforms and applications, available as open source and commercial products.



Debugging 4: VxWorks compared to Linux

- Physical/virtual address duality
- Strict access control to memory
- Difference between processes and threads
- Separate app and system code
- Driver model
- Less use of h/w debug tools, more s/w debug

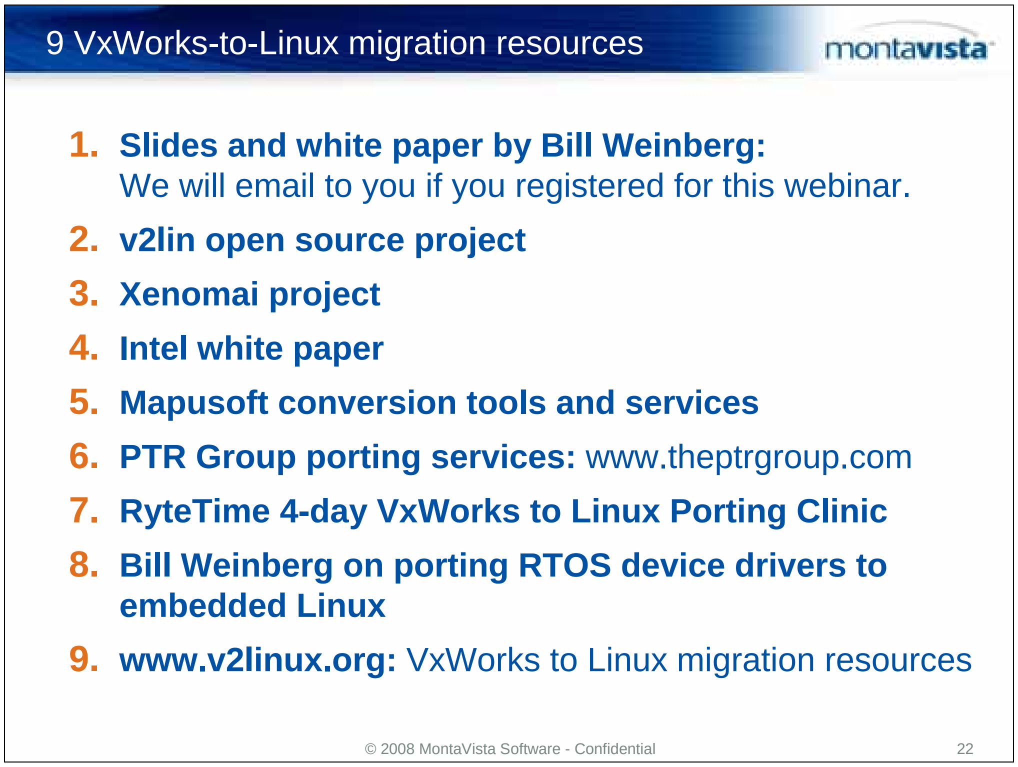
	VxWorks	Linux	Comments
BIOS/ROM	JTAG	JTAG	99% comparable. Main differences lie in use of proprietary vs. open source boot ROMs.
Kernel load/boot	JTAG	JTAG	Need to decompress and load both kernel *and* file system images. Big changes after enabling MMU .
Kernel initialization	JTAG	MMU-aware JTAG	Linux kernel runs in (contiguous) virtual address space, requiring MMU-aware JTAG h/w debug.
Kernel/drivers	JTAG	JTAG, KGDB, KDB, third party	KGDB uses serial or polled Ethernet; KDB is a target-based native debug tool.
Application debug	Various	GDB and GDBserver over IP (Ethernet, serial, USB), native/local, GDB over JTAG, third party	Usually involves a s/w debug agent. Paged code and data contiguous in logical but not physical memory. Multiple developers can debug different applications simultaneously on single target.

21

As with all types of migration, debugging legacy VxWorks applications after porting to Linux presents developers with a mix of both familiar and new processes and tools. In particular, developers migrating to Linux from VxWorks will need to become comfortable with:

- Physical/virtual address duality not present in legacy VxWorks
- Strict access control to physical memory and memory-mapped I/O
- Different rules and behaviors for debugging processes and threads (e.g., ability to attach to running processes)
- Strict separation of application and system code (kernel, drivers, modules)
- A different driver model (formal Linux drivers vs. ad hoc legacy VxWorks I/O code)
- Limited utility from traditional h/w debug tools (e.g., JTAG) and more dependence on s/w debug agents (e.g., gdbserver)

This table compares the primary debug activities you probably perform on legacy VxWorks code, and how Linux impacts those activities.



9 VxWorks-to-Linux migration resources

1. **Slides and white paper by Bill Weinberg:**
   We will email to you if you registered for this webinar.
2. **v2lin open source project**
3. **Xenomai project**
4. **Intel white paper**
5. **Mapusoft conversion tools and services**
6. **PTR Group porting services:** www.theptrgroup.com
7. **RyteTime 4-day VxWorks to Linux Porting Clinic**
8. **Bill Weinberg on porting RTOS device drivers to embedded Linux**
9. **www.v2linux.org:** VxWorks to Linux migration resources

1. Slides and white paper by Bill Weinberg: **We will email to you if you registered for this webinar.**
2. v2lin open source project: **http://v2lin.sourceforge.net/**
3. Xenomai project: **www.xenomai.org/index.php/Main_Page**
4. Intel white paper: **http://sunsite.rediris.es/pub/mirror/intel/intarch/PAPERS/30910301.pdf**
5. Mapusoft conversion tools and services: **www.mapusoft.com**
6. PTR Group porting services: **www.theptrgroup.com**
7. RyteTime 4-day VxWorks to Linux Porting Clinic: **www.rytetyme.com/c_vx2linux.html**
8. Bill Weinberg on porting RTOS device drivers to embedded Linux: **www.linuxjournal.com/article/7355**
9. New v2linux.org migration resources: **www.v2linux.org**