Using an RTOS to Implement Symmetric Multiprocessing

By Srinivas Dharmasanam

Although multiprocessor systems have been around for decades, they have only recently attained mainstream adoption. One reason is the falling cost, and increased availability, of off-the-shelf multiprocessor hardware. Another is the growing number of applications that require extremely high performance and scalability. And yet another is the increased demand for so-called “high availability” systems that can deliver 99.999% uptime. Together, these factors have spurred the development of several types of multiprocessor systems, each designed for a specific purpose. The end result: multiprocessor systems have now achieved widespread use in a variety of applications, including fault-tolerant server clusters, 3D graphics systems that perform compute-intensive audio/video compression and decompression, and high-bandwidth switch/router designs that implement network management features on specialized multiprocessors.

This paper focuses on a specific type of multiprocessor environment — the tightly coupled shared-memory symmetric multiprocessor (SMP) system — and how it is supported by a typical real-time operating system (RTOS).

Tightly Coupled vs. Loosely Coupled

In an SMP system, the term “tightly coupled” means that the individual processor cores lay close to each other, physically connected over a common high-speed bus. Furthermore, the processors share peripheral I/O devices and a global memory module (hence, “shared memory”) through a common bus interface.

It is important to distinguish an SMP system from a distributed multiprocessor environment. In a distributed multiprocessor system, the individual processing units typically reside as separate “loosely coupled” nodes, where each node can have a different processor type and its own memory and I/O devices. Moreover, each processor runs its own operating system and synchronizes with other processors using messages or semaphores over an interconnect such as Ethernet or a specialized high-speed interconnect such as Infiniband.

In comparison, a tightly coupled shared-memory SMP system runs a single copy of an operating system that coordinates simultaneous activity occurring in each of the identical CPUs. Since these processors access a common memory region, they synchronize with each other using a communication mechanism based on low-latency shared memory.

Symmetry

An SMP system, by definition, has multiple identical processors connected to a set of peripheral devices, such as memory and I/O devices on a common high-speed bus. The term "symmetric" is used because each processor has the same access mechanism to the peripheral devices.

Figure 1 illustrates an SMP system with two identical CPUs (P1 and P2), a global memory module, and I/O devices, including an Ethernet network interface card (NIC) and a disk controller.

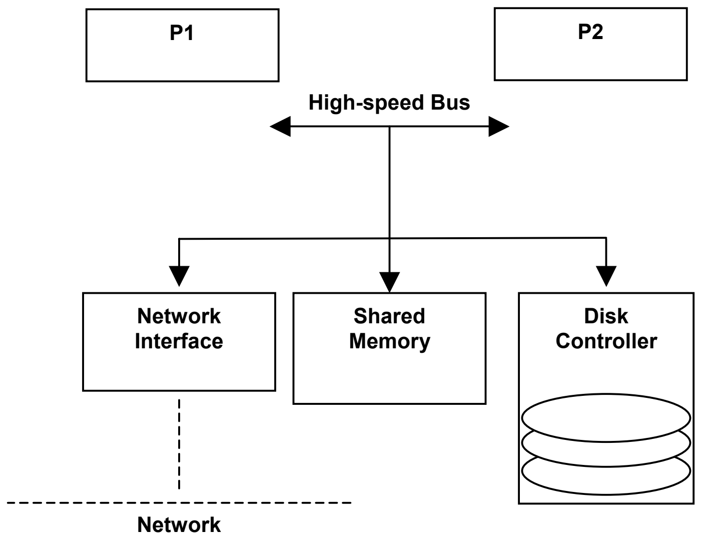


Figure 1 — Shared-memory symmetric multiprocessor architecture.

In an SMP environment, processors have to contend for access to the bus. For instance, if P1 is occupying the bus to receive data from the network interface, P2 has to wait for the operation to complete before getting access to write data to the disk. Thus, while any device is busy accessing the memory for a read-write operation, all the other devices seeking access must enter the "busy-wait" state.

Nonetheless, each device typically has a local memory reserved for its own use. For example, a NIC may use local memory to buffer and preprocess data frames received over the network before transferring the data (typically through DMA) to the global memory for further processing by P1 or P2.

Besides the per-CPU interrupt controller, an SMP system usually has another interrupt controller that routes the interrupt request signals from the peripheral devices to one of the CPUs for servicing the interrupt.

Because the hardware architecture is symmetric, the software design can remain independent of hardware-related issues, such as the specific interconnection scheme used by peripheral devices and the actual number of processing elements in the system.

Why SMP?

System designers may implement multiple processors in a symmetric fashion for a number of reasons.

Among them:

- **Greater concurrency** — Two processing units running in parallel could complete a set of tasks more quickly than one processor. Likewise, four processors could be better than two.

- **Greater scalability** — An system in which more processing cores or peripheral devices can be added to execute more tasks in parallel offers a viable platform for scalability.

The Role of the OS

Supporting SMP in an operating system is relatively straightforward. In fact, a suitably designed operating system will make the actual number of processors transparent to the application. Existing application software can, as a result, run without SMP-specific modifications.

Much like its uniprocessor equivalent, a preemptive RTOS running on an SMP system guarantees that, at any given time, tasks of the highest priority are running. The difference is that each processor can run one task, allowing multiple tasks to run concurrently. If any task of higher priority becomes ready to run, it will preempt the running task of the lowest priority.

In a multithreaded application running on a typical uniprocessor system, the RTOS synchronizes access to a global resource through the principle of mutual exclusion. This principle extends to a multiprocessor system by the addition of *spin locks*.

Spin locks

A spin lock is the most basic SMP primitive; it is used to ensure data integrity in a kernel supporting SMP. Listing 1 illustrates a simple spin-lock implementation in pseudocode.

```
SPINLOCK sp_lock = 0;

lock(&sp_lock)
{
      // Try to obtain exclusive access to the lock.
      while (swap_atomic(&sp_lock, 1) = = 1)
      {
            // Lock in use.  Try again.
      }
      // The caller now has exclusive access to the lock.
}

unlock (&sp_lock)
{
      // Make the lock available to the next user.
      swap_atomic(&sp_lock, 0);
}
```

Listing 1 — Lock-unlock for short-term mutual exclusion.

The *SPINLOCK sp_lock* in the pseudocode in Listing 1 is a kernel data structure. A task running on any one processor that acquires sp_lock, by calling the *lock()* function successfully, exits the "spin" loop therein and gains exclusive access to the shared resource it protects. When the task is done with the shared resource, it releases the lock by calling the *unlock()* function. In the meantime, all the other running tasks trying to acquire that lock will spin idly while checking for *sp_lock* to be reset to 0. Only one of the waiting tasks will be admitted successfully when it is released.

Consider the example depicted in Figure 2 (following page) to see how the spin lock works in a multiprocessor environment. First, Task 1 acquires the lock before entering its critical code section. While Task 1 is still accessing the shared resource, Task 2, which is running on a different processor, tries to acquire the same lock and then spins, waiting for Task 1 to release it. Subsequently, Task 3 also calls *lock()* and spins idly. When Task 1 completes its critical code execution and releases the lock, Task 3 acquires the lock**,** even though Task 2 attempted to acquire the lock first.

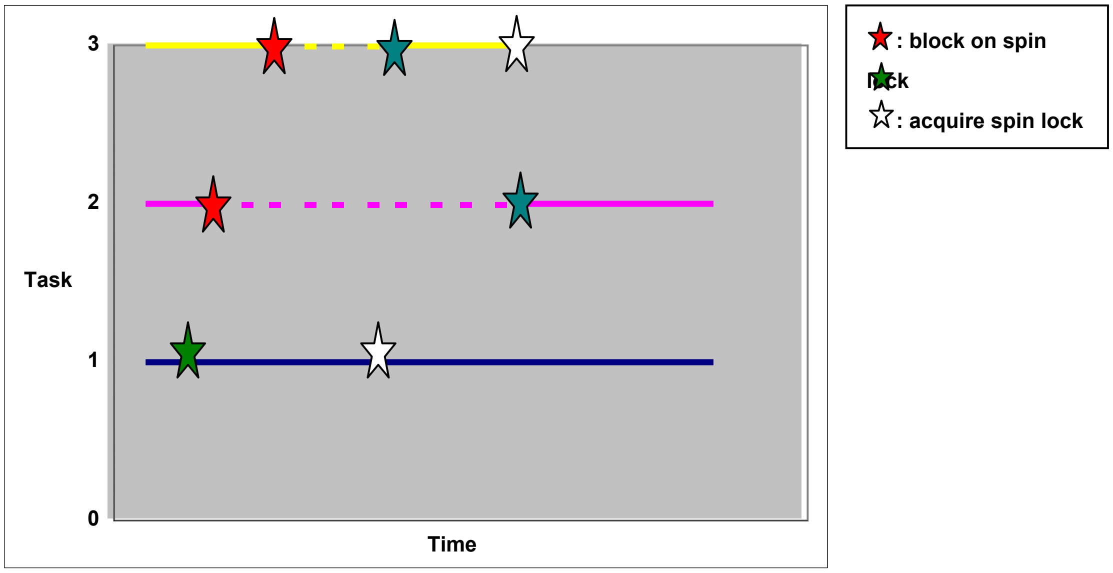


Figure 2 — Propagation of single giant lock among tasks

Note that although the tasks enter spin-wait "states," as described above, they are still "running" on one of the processors; hence there is no context switch during this sequence of operations. And since these three tasks are running at the same time, we can also conclude that system has at least three individual processing units.

Immediately after the release of the spin lock, the spinning task that first tries to acquire the lock gets it and exits the *while* loop to continue with its regular operation. If the RTOS supports prioritized spin locks [3], then, upon release of the spin lock by Task 1, the lock is given to the highest-priority task that is trying to obtain the lock.

An alternative approach is to let the task that is spinning idly sleep for a while and switch to running another user-mode task. Upon release of the spin lock, a *wakeup*() function is called that causes all the tasks sleeping on the spin lock to be woken up.

```
SPINLOCK sp_lock = 0;

lock(&sp_lock)
{
    // Try to obtain exclusive access to the lock.
    while (swap_atomic(&sp_lock, 1) = = 1)
    {
        // Lock in use.
        sleep(&sp_lock); // sleep for some time
    }
    // The caller now has exclusive access to the lock.
}

unlock (&sp_lock)
{
    // Make the lock available to the next user.
    swap_atomic(&sp_lock, 0);
    wakeup(&sp_lock); // wakeup tasks that slept for this lock
}
```

Listing 2 — Lock-unlock with sleep-wakeup for long-term mutual exclusion.

Listing 2 shows pseudocode for a spin lock-unlock implementation with associated sleep-wakeup. This implementation is better suited where long-term mutual exclusion for a resource is desired. Otherwise, if the lock is held for only very short durations of time, the context switch overhead associated with these sleep-wakeup calls could consume any performance improvement achieved in switching to other tasks instead of keeping the processor tied up with one task spinning idly for the lock, as in Listing 1.

Now that we have seen how a basic spin lock works, let us see how they are used to ensure data integrity inside a multiprocessor kernel.

Architectures

To support SMP hardware, an RTOS may use one of several possible architectures.

Master-Slave

The simplest way a uniprocessor kernel can be modified to support multiprocessor systems is to treat the entire operating system as one indivisible entity and restrict all kernel mode activity to always run on the same processor known as the “master” processor. The other processors, called “slave” processors, execute only user-mode operations and, hence, the resulting software architecture is no longer symmetric.

A task running on a slave processor may request a kernel service by entering a queue of tasks waiting to run on the master for kernel-mode execution. When the master processor is free, the task of highest priority waiting in this queue is allowed to run. Upon completion of the requested kernel service, the task, if it is still ready to run, enters a second queue for user-mode tasks for execution on one of the slaves.

Note that the two queues (one for managing tasks requesting execution on the master and the other for any slave) must be protected via spin locks. Tasks running in parallel on different processors may try to access these queues at the same time. The spin locks prevent race conditions associated with the queues.

Other kernel data structures do not require spin lock protection, since the kernel always executes on the master processor. Thus, as far as the kernel is concerned, it is still running in a uniprocessor environment. Only user-level tasks can exploit the parallelism offered by the slave processors.

A good example of such an implementation is the pSOS+m RTOS from Wind River Systems.

The drawback of the master-slave architecture is that it works well only when the tasks executes for most of the time in user mode. In applications that involve a significant percentage of kernel mode operations, a master-slave kernel serializes all such operations such for execution on the master. Hence the system as a whole performs as if it's still running on a uniprocessor system. Furthermore, if several tasks concurrently request kernel services, the overhead associated with these tasks entering and leaving the run queues can significantly impact their overall performance.

With a master-slave RTOS, if the application consists of 40% kernel mode execution, then having two CPUs instead of one can result in a best-case estimate of 30% improvement in system performance where the remaining 60% of overall CPU usage involving user-mode operation gets divided equally between the two processors.

As the kernel mode execution time increases as a percentage of the overall CPU usage, and as more tasks enter and leave the run queues as discussed above, the application performance degrades rapidly.

Giant Lock, Coarse-Grained Locking

A refinement of the master-slave kernel is the so-called *giant lock* architecture. The entire operating system is treated as one *monolithic* entity protected by one spin lock, but does not restrict kernel mode operations to one specific processor.

Any task requesting a kernel mode operation can acquire the giant lock and continue to run on the same processor. However, kernel mode execution is still restricted to any one of the processors at any given time. So while one task holds the giant lock, all others waiting to use kernel services will idle.

Since there is no notion here of a master or a slave processor — true to the "symmetric" hardware architecture of the SMP system — this RTOS design allows any processor to execute kernel mode operations. Although the overhead of switching to a different processor is eliminated with this architecture, note that the task waiting for the lock will spin idly waiting for the lock, unlike a master-slave kernel where the processor can switch to another user mode task while the current task waits in the master queue.

The example illustration in Figure 2 is one such design where the entire kernel is treated as one entity and protected by *a* single spin lock. Notice that Task 2 spends a significant amount of time spinning idly, waiting for the giant lock to be released by Task 3. It is possible, therefore, that if Task 1 then Task 3 occupy the lock for extended periods of time, Task 2 will continue to spin idly and never be able to do anything useful — although it is still "running" on one of the processors. Hence, if several such tasks are waiting for a kernel service, then these tasks will execute one after the other, serially, as if on a uniprocessor system, and not together concurrently, as desired when we opted for a multiple processor system.

This is the essential drawback of the giant-lock kernel architecture. Having multiple processors may not actually improve the application performance if several tasks need kernel mode operation for significant amounts of time. As a result of this poor "scaling" capability and the adverse impact it has on interrupt latencies, this design is not used in most real-time operating systems that support multiple processors.

An example of such a kernel lock implementation is the FreeBSD 4.2 operating system, which has a traditional non-real-time UNIX kernel.

The giant lock kernel is one extreme of a coarse-grained kernel lock design. To provide good scaling behavior, an SMP operating system needs to have separate locks for different kernel subsystems. So-called fine-grained kernel locking is designed to allow individual kernel subsystems to run as separate threads on different processors at the same time — thus allowing a greater degree of parallelism among the tasks in an application.

Fine-Grained Locking

The goal underlying a fine-grained kernel locking architecture is to allow tasks running on different processors to concurrently execute kernel-mode operations. The result is often called a threaded kernel. This goal is accomplished by using separate spin locks for different kernel subsystems, so that tasks trying to access these individual subsystems can run concurrently.

The "granularity" of the locking mechanism determines the maximum number of concurrent kernel threads. For example, the core scheduler and file system components of most operating systems are relatively independent kernel services. Hence, two different spin locks can protect these two subsystems. Thus, a task performing a large file read/write involving time-consuming disk I/O does not have to block another task attempting to access the scheduler to, say, activate another task. Hence, having separate spin locks for these two subsystems results in a "threaded" kernel, where two different tasks could execute in kernel mode at the same time.

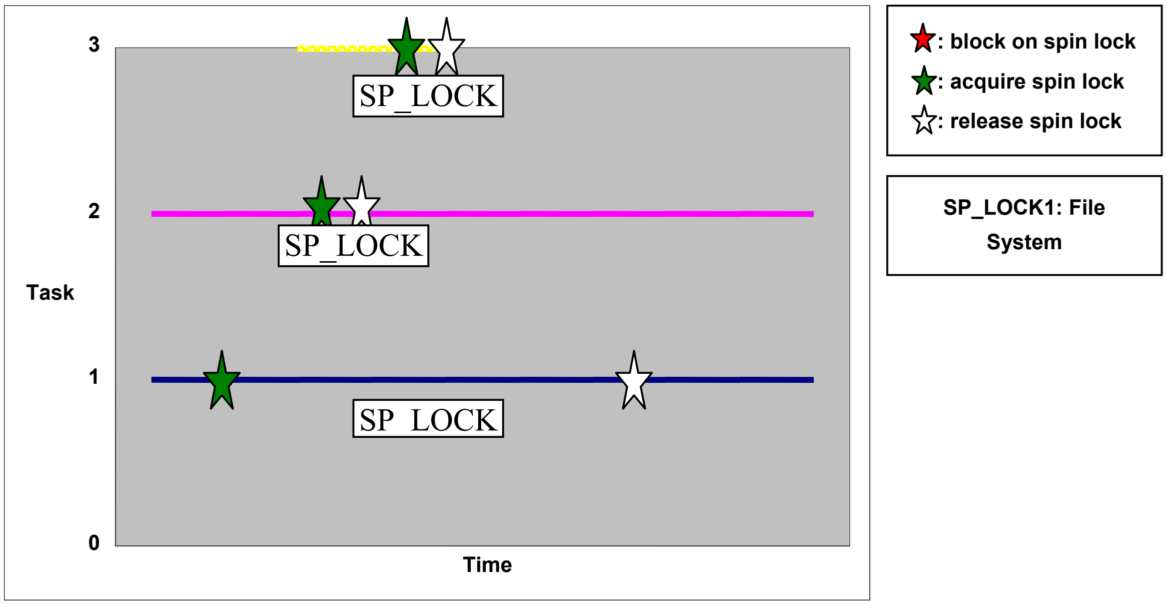


Figure 3 — Two separate spin locks

As shown in Figure 3, while Task 1 is busy doing disk I/O, Task 2 can activate the high-priority Task 3 — and thereby allow Task 2 and Task 3 to do useful operations instead of spinning idly. Notice that the spin lock acquired by Task 3 is released before that task disables itself. Otherwise, any other task trying to acquire the same lock would spin idly forever.

A fine-grained kernel locking mechanism can enable a much greater degree of parallelism in execution of user tasks. This helps boost CPU utilization and the overall application throughput.

Nonetheless, having additional spin locks for different kernel subsystems introduces the overhead of having to manage each spin lock separately.

Using multiple spin locks also introduces a potential for deadlock. For example, in the scenario depicted in Figure 3, an alternative outcome is:

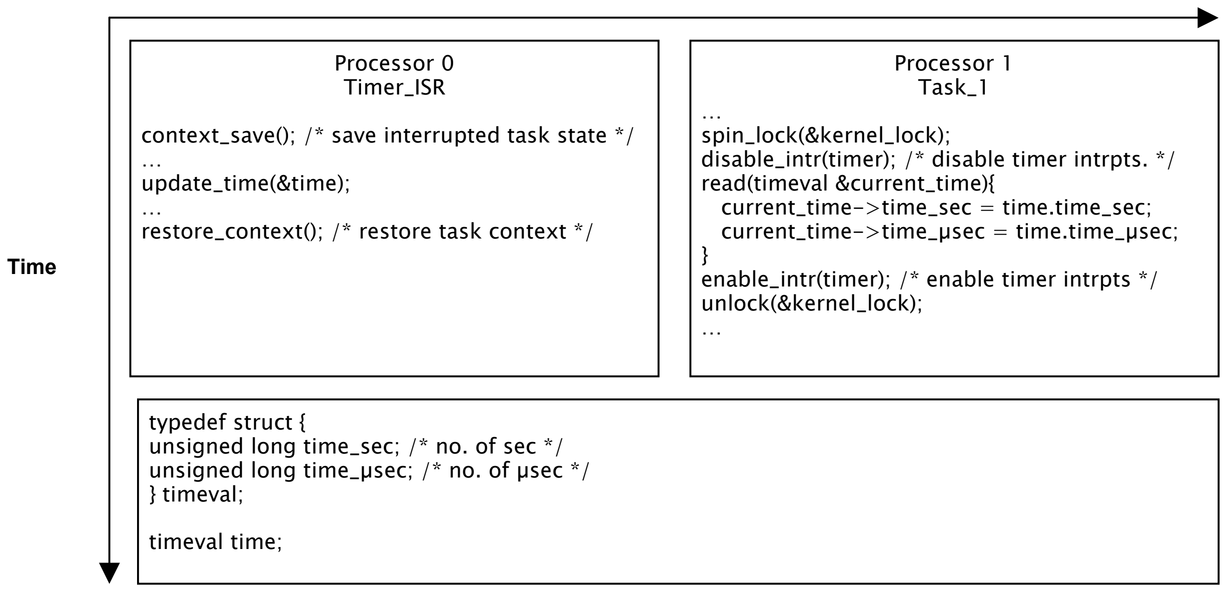
Task 1 acquires the SP_LOCK1 for disk I/O.

Task 2 acquires the SP_LOCK2, the scheduler lock.

Task 2 then tries to acquire SP_LOCK1 and spins, waiting for Task 1 to release SP_LOCK1.

Task 1 then tries to acquire SP_LOCK2 and spins, waiting for Task 2 to release SP_LOCK2.

This results in a deadlock that causes the entire system to hang. The problem gets more complicated if there are several such spin locks in the operating system. (This problem can be addressed by implementing the so-called prioritized spin locks [3] with priority ceiling or priority inheritance protocols).



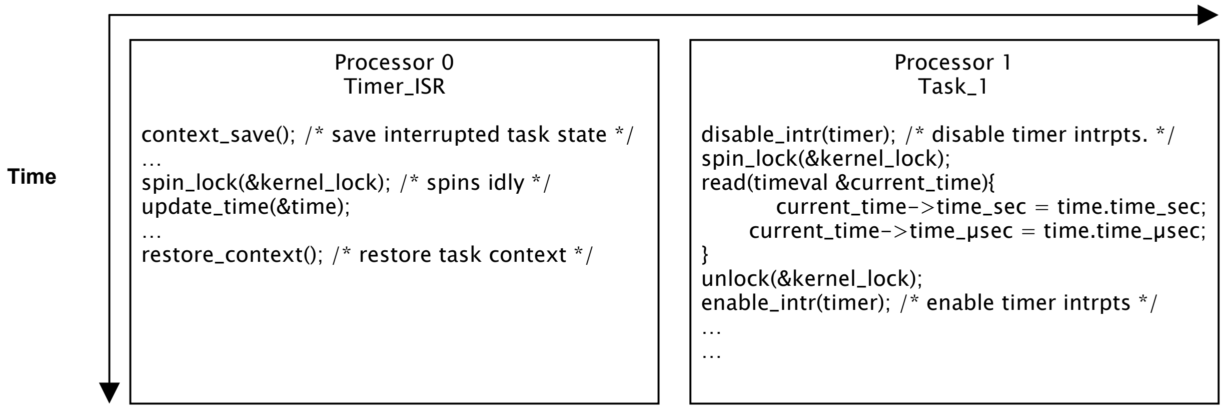
```
context_save(); /* save interrupted task state */
…
update_time(&time);
…
restore_context(); /* restore task context */
```

```
…
spin_lock(&kernel_lock);
disable_intr(timer); /* disable timer intrpts. */
read(timeval &current_time){
   current_time->time_sec = time.time_sec;
   current_time->time_μsec = time.time_μsec;
}
enable_intr(timer); /* enable timer intrpts */
unlock(&kernel_lock);
…
```

```
typedef struct {
unsigned long time_sec; /* no. of sec */
unsigned long time_μsec; /* no. of μsec */
} timeval;

timeval time;
```

Listing 3 – A multiprocessor design with a race condition



```
context_save(); /* save interrupted task state */
…
spin_lock(&kernel_lock); /* spins idly */
update_time(&time);
…
restore_context(); /* restore task context */
```

```
disable_intr(timer); /* disable timer intrpts. */
spin_lock(&kernel_lock);
read(timeval &current_time){
        current_time->time_sec = time.time_sec;
     current_time->time_μsec = time.time_μsec;
}
unlock(&kernel_lock);
enable_intr(timer); /* enable timer intrpts */
…
…
```

Listing 4 – A version that lacks the race condition

Thus, the overhead of managing multiple spin locks must be balanced carefully with the desired degree of parallelism in the application. The critical aspect of this design is identification of the kernel subsystems that require separate spin locks. By carefully selecting a set of kernel subsystems to be protected by separate spin locks, a typical RTOS design can provide good scaling behavior of its SMP implementation for certain types of applications.

The Embedded Linux RTOS from MontaVista, a derivative of the Linux 2.4 with the fully preemptive scheduler, has a fine-grained locking mechanism inside the SMP kernel. This design allows user tasks using these services to run concurrently as separate kernel mode threads on different processors.

True SMP

Among the more popular commercial RTOS vendors, QNX Software Systems and Wind River offer forms of multiprocessor support. The multiprocessor extension for Wind River's VxWorks uses synchronization primitives based on a message-passing interface between distinct VxWorks sessions running on each node. This approach is targeted more towards the multi-board designs with board-level interconnects, rather than a traditional SMP design.

The QNX Neutrino RTOS, on the other hand, supports a true SMP design based on a microkernel OS architecture rather than the monolithic architecture used by many other operating systems. As a microkernel OS, QNX Neutrino supports a core set of services, such as scheduling and interrupt management, in the kernel itself; all the other services, such as device drivers and network I/O (including the protocol stacks), run as separate user-space processes. As a result, kernel mode operations run for very short periods of time, allowing the microkernel to be treated as a single entity in implementing the SMP locking mechanism. For a detailed discussion of QNX Neutrino and its microkernel architecture, the reader is referred to the QNX Neutrino System Architecture manual.

Interrupt Management

An important aspect of RTOS design that was intentionally left out in the discussion so far is the mechanism for servicing the various interrupt-driven devices, such as the system timer and I/O devices. Interrupt latency— the time required to service an interrupt from the moment the hardware interrupt signal is asserted by a device — affects overall application performance in terms of its responsiveness to various external "events" and internal "alarms." In the case of the system timer, the interrupt latency of system clock interrupts is directly related to the scheduling latency that determines the real-time properties of the kernel.

In a typical uniprocessor RTOS, potential race conditions between tasks and interrupt handlers that access the same global shared resource are prevented by having the task temporarily disable interrupts while it executes in the critical code section. In a multiprocessor system, however, disabling interrupts isn't enough to prevent such race conditions.

Consider a situation where the interrupt handler timer_ISR (), shown in Listing 1, is being serviced and Task 1, running on another processor, disables interrupts and enters its critical code section. Here we have the system clock interrupt handler, *timer_ISR*(), running concurrently with Task 1, both accessing the global data *time* that holds the current time value. Thus, we have a race condition between Task 1 and timer_ISR () where the current time in seconds, read by Task 1, does not correspond with the current time in microseconds, *current_time.time_µsec*.

To avoid race conditions between tasks and interrupt service routines in an SMP system, both the tasks and the ISRs must gain exclusive access to the shared resource by obtaining a spin lock for the resource used by their respective critical code sections.

Modified pseudocode for Task 1 and timer_ISR () is shown in Listing 2. If the ISR finds that the spin lock is already in use, then it too will spin idly, waiting for the task or another ISR to release it. This will affect interrupt latency and overall system responsiveness to external events.

In the master-slave configuration, this type of race condition is implicitly avoided, since all interrupt handlers, being kernel mode operations, are restricted to run on the master. In the giant lock and fine-grained kernel lock designs, however, we need to implement this spin lock protection in the interrupt handler routines.

The resulting degradation of interrupt latency is especially bad with the giant-lock architecture since all tasks and ISRs must vie for access to the single kernel spin lock.

Typically, if an application is I/O-intensive, with a high degree of interrupt-driven data transfer involving kernel mode operations, fine-grained kernel locking is essential to accomplish a reasonable degree of parallelism on an SMP system.

Summary

Because of their unique capabilities, tightly coupled shared-memory SMP systems can replace existing uniprocessor systems to provide scalable platforms for CPU-intensive, high-availability designs.

Extending a uniprocessor-based RTOS to support an SMP system is relatively straightforward. This is typically accomplished adding basic primitives to the operating system, such as the spin locks. With spin locks and an appropriate kernel architecture, user-level tasks can safely and efficiently compete for shared resources while running on parallel processors. The number of spin locks should be determined according to the desired degree of parallelism in the application and the overhead of managing the locks correctly.

Migration from a uniprocessor system to an SMP system may not always result in higher overall throughput. In many cases, however, multi-threaded applications and parallelizable algorithms can achieve significant improvement.

Biography

Srinivas Dharmasanam is a principal engineer at Pillar Data Systems. He holds a Masters of Engineering degree from the University of Michigan, Ann Arbor, specializing in control systems. He can be contacted by e-mail at the_srinivas@hotmail.com.

References

1. Curt Schimmel, “UNIX Systems for Modern Architectures,” Addison-Wesley, 1994.

2. “System Architecture,” QNX Neutrino real-time operating system, QNX Software Systems Ltd.

3. T. Johnson and K. Harathi, “A prioritized multiprocessor spin lock,” Tech. Rep. TR-93-005, Department of Computer Science, University of Florida, 1993.