Pulling the threads together

How concurrency checkers can improve multicore process performance. By **Paul Anderson**.

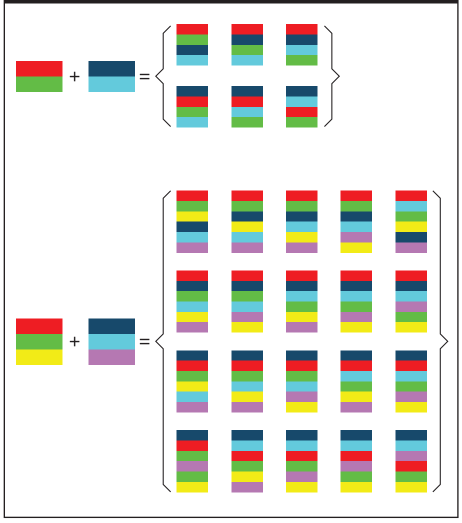
In order to take advantage of the performance improvements provided by multicore processors, a good command of concurrency is essential. Yet most developers reason from the perspective of a single thread of execution and may use techniques and tools that are themselves fundamentally single threaded.

Concurrency opens up programs to entirely new classes of defects, such as starvation, deadlocks and race conditions. At the same time, the non determinism and the sheer number of possibilities introduced by thread interleaving make it significantly more difficult to find bugs in multithreaded systems by testing and other traditional methods.

One of the greatest strengths of concurrent execution is also one of the biggest sources of problems: instructions in multiple threads can be interleaved. The number of possible interleavings increases enormously as the number of instructions grows; a phenomenon known as the combinatorial explosion. If thread A executes M instructions and thread B executes N instructions, there are $^{N+M}C_N$ possible interleavings of the two threads. Even the smallest threads have many possible interleavings. Figure 1 shows the possible interleavings of two threads containing two instructions or three instructions each; with 12 instructions each, the number of possible interleavings exceeds 2million.

Real concurrent programs have astronomical numbers of legal interleavings, so testing every interleaving is infeasible. This would not matter if all interleavings were equivalent; in general, they are not, and interleaved threads can affect each other's behaviour. Ideally, these effects are intentional and correct; in practice, they may not be. Concurrent programs that run perfectly well on single processor systems often manifest previously latent defects when run on multiprocessor systems. Advanced static analysis tools thus play an increasingly important role. They use sophisticated symbolic execution techniques to reason about many possible execution paths and interleavings at once. These techniques can find concurrency errors without needing to execute the program at all.

Fig 1: Possible interleavings of multiple threads



Race conditions

One of the most common unintended consequences of thread interleaving is the race condition – a class of problem that does not exist in a single threaded world. A race condition arises when multiple threads of execution access a shared piece of data – at least one of them changing its value – without an explicit synchronisation operation to separate the accesses. Depending on the thread interleaving, the system can be left in an inconsistent state.

A simple example is shown in fig 2. A manufacturing assembly line maintains a running count of items completed, with separate controllers

Fig 2: A race condition can lead to an incorrect count of items on an assembly line.

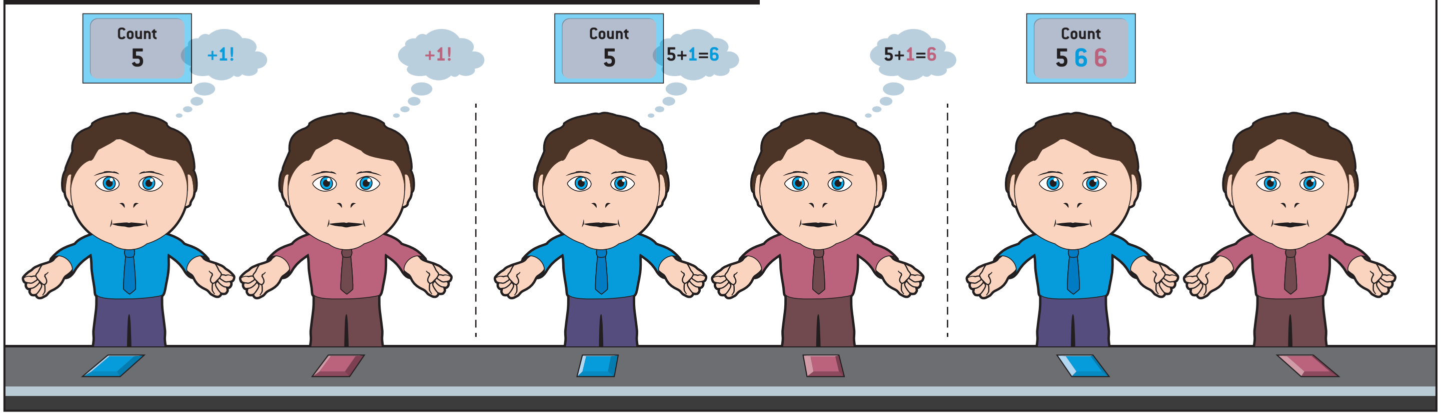
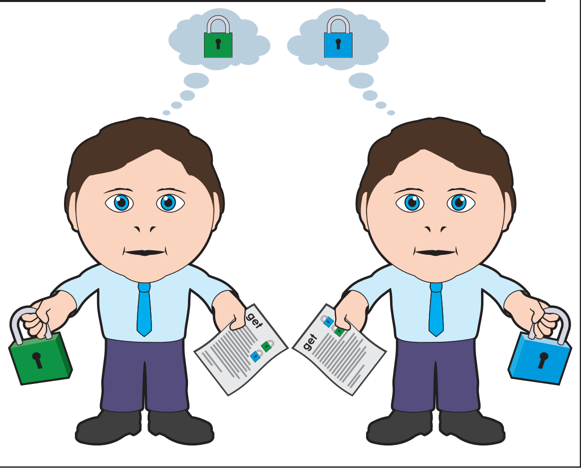


Fig 3: Deadlock between two threads: neither can progress



responsible for counting each kind of object. In a multithreaded system, a race condition can arise because the controllers read and write a shared piece of data: the count. Many interleavings will result in correct counts but – critically – some do not.

Several factors make race conditions difficult to eliminate. Firstly, the problem may not show up during testing, especially if only a small proportion of the possible interleavings lead to inconsistent outcomes. Secondly, race condition diagnosis is difficult. Programmers accustomed to a single threaded world may not immediately consider the possibility of a race condition. The symptoms can be perplexing; in fig 2, for example, the running count will usually be correct, but sometimes too low. This effect can seem impossible when the code is considered in isolation, which sometimes leads developers to discard race related bug reports as unreproducible. Static analysis tools typically identify race conditions by examining accesses to shared memory locations – they focus on reasoning, rather than test executions, and causes, rather than symptoms.

Anderson: "Concurrency opens up programs to entirely new classes of defects."

Race conditions and the ensuing errors are common, even in (well tested) deployed software and their consequences can be devastating. The 2003 blackout in the north east of the US was worsened by a race condition that caused delayed and misleading information to be communicated by a computerised energy management system. Kevin Paulsen, writing in *Security Focus*, noted 'the bug had a window of opportunity measured in milliseconds'. The chances that a problem like this could have manifested itself during testing are infinitesimal.

Out of the frying pan

Extensive efforts have been made towards developing techniques to protect shared resources and to eliminate race conditions. Unfortunately, these techniques introduce problems of their own, including performance bottlenecks and increased code complexity. In the worst case, they can introduce deadlock and starvation.

In a deadlock, two or more threads prevent each other from making progress by each holding a lock needed by another. Figure 3 shows a deadlock between two threads that both require the same two locks and have managed to obtain one each. Neither thread can obtain the second lock it needs, so neither can carry out its operations: both threads are completely stuck.

Deadlock can only arise if different threads try to acquire the same locks in different orders. Because this property can be detected statically, risk of deadlock can be detected, even if no deadlocks have emerged in testing. A more aggressive approach is to forbid any thread from holding more than one lock at a time: a property that is also statically checkable.

Even with deadlock eliminated, process starvation can still occur. A thread holding a lock for a long time – for example while waiting for a large data transfer – causes other threads requiring that lock to starve. In the worst case, some or all of the other threads may never have the opportunity to run. Figure 4 shows one long running thread causing starvation in three others.

Static analysis can help by identifying calls to long running functions – such as sleep() – that occur while a lock is held. Various static tools have built in checks of this form. Some, such as CodeSonar, allow functions to be identified as 'long running' for the sake of such checks.

Static analysis tools are also well suited to enforcing coding standards related to synchronisation. For example, standards may require that a lock and its corresponding unlock take place in the same function, or that the lock referred to by such an operation be statically identifiable. Such rules can reduce the incidence of runtime problems, as well as making code easier to read and maintain.

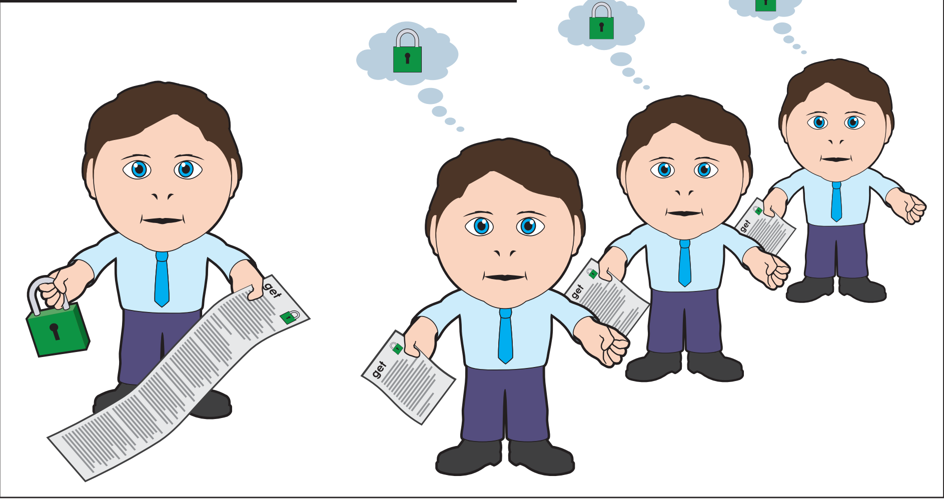
Conclusion

More cores can lead to major performance gains, but also to major development challenges and the potential for bugs with major consequences. Static analysis provides important leverage in the multithreaded development lifecycle and is strongly recommended as an adjunct to traditional testing and debugging activities.

Author profile:

Paul Anderson is vice president of engineering for GrammaTech. **www.grammatech.com**

Fig 4: A long running thread may cause others to starve


get
get
get
get