Having discussed how to keep data coherent, let us now consider how synchronization is managed in bus-based multiprocessors.

5.5 SYNCHRONIZATION

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations: mutual exclusion, point-to-point events, and global events. There has been considerable debate over the years about how much hardware support and exactly what hardware primitives should be provided to support these synchronization operations. The conclusions have changed from time to time with changes in technology and design style. Hardware support has the advantage of speed, but moving functionality to software has the advantages of cost, flexibility, and adaptability to different situations. The classic works of Dijkstra (1965) and Knuth (1966) show that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization methods rely on hardware support for some sort of *atomic read-modify-write* operation, in which the value of a memory location is ensured to be read, modified, and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built in software using these primitives.

The history of instruction sets offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare&swap* instruction, to support synchronization in concurrent programming on uniprocessor or multiprocessor systems. The compare&swap compares the value in a memory location with the value in a specified register and, if they are equal, swaps the value in the memory location with the value in a second specified register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic; since the source and destination operands are memory locations, much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high-level language architecture have proposed that the user-level synchronization operations, such as locks and barriers, should be supported directly at the machine level, not just atomic read-modify-write primitives; that is, the synchronization "algorithm" itself should be implemented in hardware. This issue became very active during the reduced instruction set debates since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The Sparc approach was to provide atomic operations involving a register or registers and a memory location using a simple swap (atomically swapping the contents of the specified register and memory location) and a compare&swap. MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described later in this section, which allows a variety of higher-level read-modify-write operations to be constructed without requiring the design to implement them all. In essence, the pair of instructions can be used instead of a single instruction to implement atomic exchange or more complex atomic operations. This approach was later incorporated into the PowerPC and DEC Alpha architectures and is now quite popular. As we will see, synchronization brings to light a rich family of trade-offs across the layers of communication architecture. Not only can a spectrum of high-level operations and low-level primitives be supported by hardware, but the synchronization requirements of applications vary substantially as well.

The focus of this section is on how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor through a combination of software algorithms and hardware primitives. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high-level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Then, given that the hardware supports only the basic atomic operations, we can examine the role of the user software and system software in synchronization operations and then consider the hardware and software design trade-offs in greater detail.

5.5.1 Components of a Synchronization Event

There are three major components of a synchronization event:

1. *Acquire method:* a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization).
2. *Waiting algorithm:* a method by which a process waits for a synchronization to become available; for example, if a process tries to acquire a lock but the lock is not free, or to proceed past an event but the event has not yet occurred.
3. *Release method:* a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the Unlock operation, a method for the last process arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization. There are two main choices: busy-waiting and blocking. *Busy-waiting* means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the waiting process to proceed. Under *blocking,* the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awakened and made ready to run again when the release it was waiting for occurs. The trade-offs between busy-waiting and blocking are clear. Blocking has higher overhead since suspending and resuming a process involves the operating system (and suspending and resuming a thread involves the run-time system of a threads package), but it makes the processor available to other threads or processes that have useful work to do. Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting. Blocking is strictly more powerful than busy-waiting because, if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.[7] Busy-waiting is likely to be better when the waiting period is short, whereas blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, the process blocks, allowing other processes to run (a *two-phase* waiting algorithm).

The difficulty in implementing high-level synchronization operations in hardware is not the acquire or the release component but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release methods and allow the three components to be glued together in software. However, subtle but very important hardware/software interactions remain in how the spinning operation in the busy-wait component is realized.

5.5.2 Role of the User and System

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher-level operations without having to worry about their internal implementation. The implementation is left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed that approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are very attractive. As with other aspects of system design, the utility of faster operations with more hardware support depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different run-time conditions. For example, a lock may be accessed with low contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements.

7. This problem of denying resources to the critical process or thread is one that is actually made simpler with more processors. When the processes are time-shared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Multiprogramming environments on a limited set of processors may fall somewhere in between.

Under high contention, most processes will spend time waiting, and the key requirement of a lock algorithm is that it provide high lock-unlock transfer bandwidth; under low contention, the key goal is to provide low latency for lock acquisition. Different algorithms may satisfy different requirements better, so we must either find a good compromise algorithm or provide different algorithms for each type of synchronization from which a user can choose. If we are lucky, a flexible library can at run time choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic hardware primitives, so some may be better suited to a particular machine than others. Under multiprogramming, process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.5.3 Mutual Exclusion

Mutual exclusion (lock-unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low-contention case. After a brief discussion of hardware locks, this section describes the simplest software algorithms for memory-based locks using atomic exchange instructions. Following this is a discussion of how these simple algorithms can be implemented by using the special load-locked and store-conditional instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the tradeoffs are. Next, we will look at more sophisticated algorithms that can be built using either method of implementing atomic operations.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used on some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines which processor gets the lock next when there are multiple requestors. However, this approach was quite inflexible since only a limited number of locks can be in use at a time and the waiting algorithm is fixed (typically a form of busy-wait with abort after time-out). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software locks in memory. The CRAY Xmp provided an interesting variant of this approach. A set of registers was shared among the processors, including a fixed collection of lock registers. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general-purpose setting, and in practice the lock registers too were used primarily to implement higher-level locks in memory.

Simple Software Lock Algorithms

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and, if it is, then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock acquire operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1, thus marking the lock busy; if the variable is 1 (lock is busy), then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). The following are assembly-level instructions for this attempt at a lock and unlock. (In our pseudo-assembly notation, the first operand always specifies the destination if there is one.)

```
lock:  ld   register, location /*copy location to register*/
       cmp  register, #0       /*compare with 0*/
       bnz  lock               /*if not 0, try again*/
       st   location, #1       /*store 1 into location to mark it locked*/
       ret                     /*return control to caller of lock*/
```

and

```
unlock:st location, #0         /*write 0 to location*/
        ret                    /*return control to caller*/
```

The problem with this lock, which is supposed to provide atomicity for the critical section that follows it, is that it needs (but lacks) atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0 and two processes P_0 and P_1 execute the above assembly code implementations of the lock operation. Process P_0 reads the value of the lock variable as 0 and thinks it is free, so it proceeds past the branch instruction. Its next step is to set the variable to 1, marking the lock as busy, but before it can do this, process P_1 reads the variable as 0, thinks the lock is free, and passes the branch instruction too. We now have two processes simultaneously proceeding past the lock and entering the same critical section, which is exactly what the lock was meant to avoid. Putting the store instruction just after the load instruction would not help either. The two-instruction sequence—reading (testing) the lock variable to check its state and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations in different processes from being interleaved in time. What we need is a way to atomically test the value of a variable and set it to another value if the test succeeds (i.e., to atomically read and then conditionally modify a memory location) and then to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported by the operating system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this requires hardware locks and tends to be slow on modern processors.

An efficient, general-purpose solution to the lock problem is to support an atomic read-modify-write instruction in the processor's instruction set. A typical approach is to have an atomic exchange instruction: a value at a memory location specified by the instruction is read into a register, and another value is stored into the location, all in an atomic operation with no other accesses to that location allowed to intervene. Many variants of this operation exist with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically. The success of the test&set is determined by examining the value in the register. If it is 0, the test&set was successful. If it is 1, it was not successful; the value 1 written to memory by the test&set instruction is the same as was already there, so no harm is done. (1 and 0 are the values typically used, though any other constants might be used in their place.) Given such an instruction, with the mnemonic t&s, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock:  t&s register, location
                                 /*copy location to reg, and set location to 1*/
       bnz register, lock        /*compare old value returned with 0*/
                                 /*if not 0, i.e., lock already busy, so try again*/
       ret                       /*return control to caller of lock*/
```

and

```
unlock:st location, #0           /*write 0 to location*/
       ret                       /*return control to caller*/
```

The lock implementation keeps trying to acquire the lock using test&set instructions until the test&set leaves zero in the register, indicating that the lock was free when tested (in which case the test&set has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic test&set instruction.

More sophisticated variants of such atomic instructions exist and, as we will see, are used by different software synchronization algorithms. One example is a *swap* instruction. Like a test&set, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location, it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly, we can implement a lock as before by replacing the test&set with a swap instruction as long as we use the values 0 and 1 and ensure that the value in the register is 1 before the swap instruction is executed; the lock has succeeded if the value left in the register by the swap instruction is 0.

Another example is the family of *fetch&op* instructions. A fetch&op instruction also specifies a location and a register. It atomically reads the current value of the location into the register and writes the value (which has been obtained by applying the operation specified by the fetch&op instruction to the current value of the location) into the location. The simplest forms of fetch&op to implement are the *fetch&increment* and *fetch&decrement* instructions, which change the current value by 1. A *fetch&add* would take another operand, which is a register or value, to add into the previous value of the location. A more complex primitive is the *compare&swap* operation. It takes two register operands and a memory location (i.e., it is a three-operand instruction, not commonly supported by RISC architectures); it compares the value in the location with the contents of the first register operand, and, if the two are equal, it swaps the contents of the memory location with the contents of the second register.

Performance of the Simple Lock

Figure 5.29 shows the performance of a simple test&set lock on the SGI Challenge.[8] Performance is measured for the following microbenchmark executed repeatedly in a loop:

```
lock(L);
critical-section(c);
unlock(L);
```

where c is a delay parameter that determines the size of the critical section (it is only a delay in this case, with no real work done). The benchmark is configured so that the same total number of lock calls are executed as the number of processors increases, reflecting a situation where a fixed number of tasks must be dequeued from a centralized task queue, independent of the number of processors. Performance is measured as the time per lock transfer, that is, the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The cumulative time spent in the critical section itself (i.e., c times the number of successful locks executed) is subtracted from the cumulative execution time so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

8. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set-based locks on older machines like the Sequent Symmetry can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991).

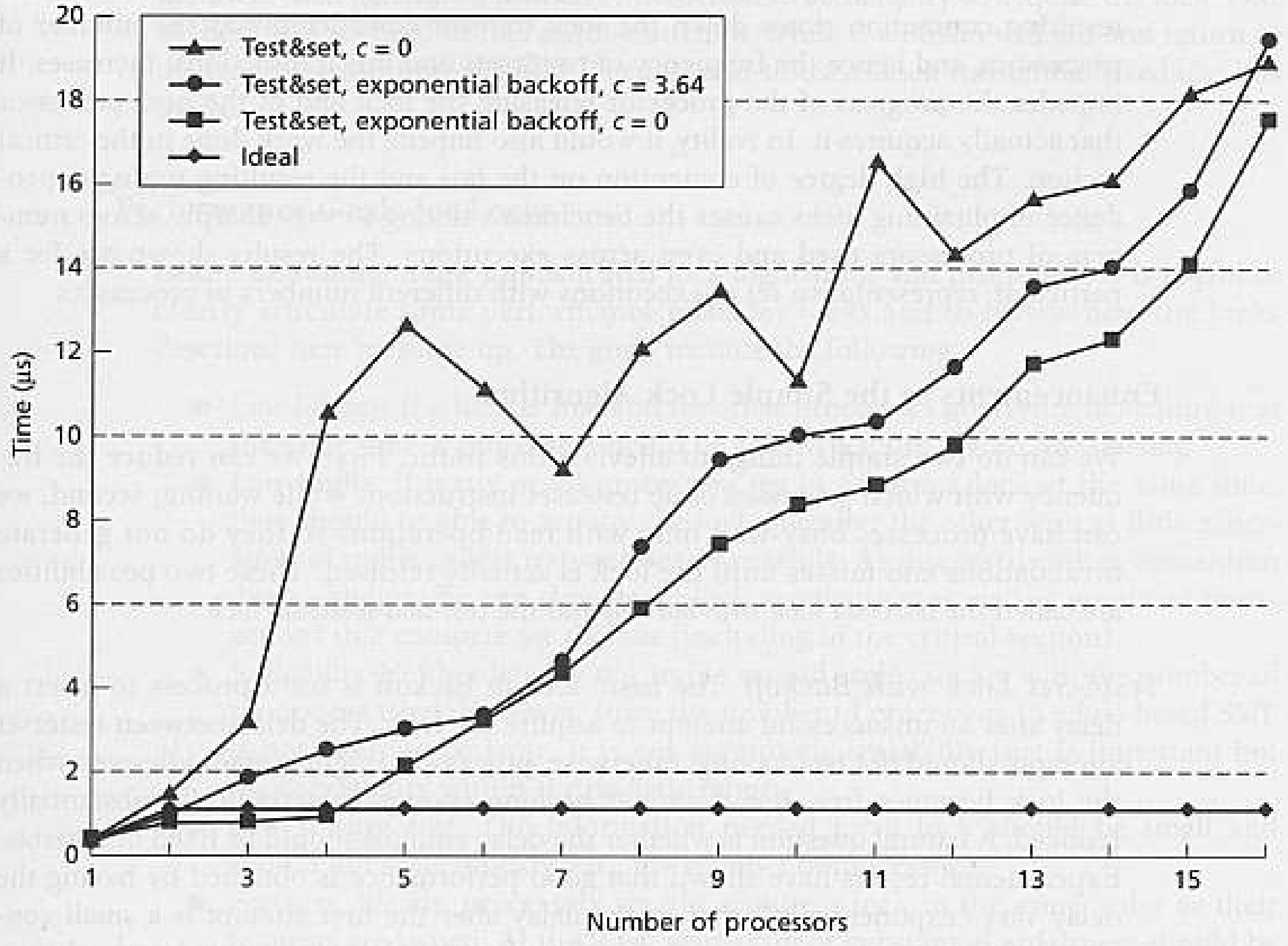

FIGURE 5.29 Performance of the synthesized test&set locks with an increasing number of competing processors on the SGI Challenge. The *y*-axis is the time per lock-unlock pair, excluding the critical section of size *c* microseconds. The irregular nature of the top curve is due to the timing dependence of the contention effects caused.

The upper curve in the figure shows the time per lock transfer with an increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with "backoff" in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labeled "ideal." However, the figure shows that performance clearly degrades with an increasing number of processors.

The problem with the test&set lock is the traffic generated during the waiting method: every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (since it uses a test&set operation and writes the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeatedly and consume precious bus bandwidth even during the waiting algorithm. The resulting contention slows down the lock transfer considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. It impedes the progress of the processor releasing the lock and of the next processor that actually acquires it. In reality, it would also impede the work done in the critical section. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

Enhancements to the Simple Lock Algorithm

We can do two simple things to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. These two possibilities are called the *test&set lock with backoff* and the *test-and-test&set lock*.

Test&Set Lock with Backoff The basic idea in backoff is for a process to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long; otherwise, processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or variable. Experimental results have shown that good performance is obtained by having the delay vary "exponentially"; that is, the delay after the first attempt is a small constant k that increases geometrically, so that after the ith attempt, it is $k \times c^i$, where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5.29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section, using the starting value k for backoff that appears to perform best. Performance improves but still does not scale very well since there is still substantial traffic interfering with the release and acquire. Performance results using backoff with a real test&set instruction on older machines can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991). See also Exercise 5.14, which discusses why the performance with a nonzero critical section is worse than that with a null critical section when backoff is used.

Test-and-Test&Set Lock A more subtle change to the algorithm is to have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly reading with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, without generating bus traffic, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available and only then will each generate a test&set instruction to actually try to acquire the lock. One of them will succeed in this acquire attempt, while the others will fail and return to the read-based waiting method. The test-and-test&set lock substantially reduces bus traffic.

Performance Goals for Locks

Before examining more sophisticated lock algorithms and primitives, it is useful to clearly articulate some performance goals for locks and to review how the locks described here measure up. The goals include the following:

- *Low latency.* If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.
- *Low traffic.* If many or all processors try to acquire a lock at the same time, they should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. As discussed earlier, contention due to high traffic can slow down lock acquisitions as well as unrelated transactions that compete for the bus (including in the critical section).
- *Scalability.* Neither latency nor traffic should scale quickly with the number of processors used. However, since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important but only scalability within the realistic range.
- *Low storage cost.* The information needed for a lock should be small and should not scale quickly with the number of processors.
- *Fairness.* Ideally, processors should acquire a lock in the same order as their requests are issued. At the least, starvation or substantial unfairness should be avoided. Since starvation is usually unlikely, the importance of fairness must be traded off with its impact on performance.

Consider the simple atomic exchange or test&set lock. It is very low latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, we have seen that it can generate a lot of bus traffic and contention if many processors compete for the lock. The performance of the lock scales poorly as the number of competing processors increases. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic, is somewhat more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended latency than the simple test&set lock (it does a read in addition to a test&set even when there is no competition) but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. (Exercise 5.12 asks you to count the number of bus transactions and the time required for the test-and-test&set type of lock in different scenarios.)

In the test-and-test&set lock, since a test&set operation (and hence a bus transaction) is only issued when a processor is notified that the lock is ready, and thereafter if it fails it busy-waits (spins) on a cached block, there is no need for backoff. However, the lock does have the problem that when the lock is released, all waiting processes rush out and perform their read misses and their test&set instructions at about the same time. The bus transactions for the read misses may be combined in a smart bus protocol; however, each of the test&set instructions itself generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case. While test-and-test&set was a major step forward at its time, better hardware primitives and better algorithms have been designed to alleviate its traffic problem.

Improved Hardware Primitives: Load-Locked, Store-Conditional

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, we would prefer that failed attempts to complete the read-modify-write do not generate invalidations. It would also be useful to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implementing each with a separate instruction. One way to achieve both goals, increasingly supported in modern microprocessors, is to use a pair of special instructions rather than a single read-write-modify instruction to implement atomic access to a variable (let's call it a synchronization variable). The first instruction, commonly called *load-locked* or *load-linked* (LL), loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register—that is, the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional*. It tries to write the register back to the memory location (the synchronization variable) if and only if no other processor has written to that location (or cache block) since this processor completed its LL. Thus, if the store-conditional succeeds, it means that the load-locked, store-conditional (LL-SC) pair has read, perhaps modified in between, and written back the variable atomically. If the store-conditional detects that an intervening write has occurred to the variable or cache block, it fails and does not even try to write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the store-conditional is indicated by the condition codes or a return value. How the LL and store-conditional are actually implemented will be discussed later; for now, we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, the simple lock and unlock algorithms can be written as follows, where `reg1` is the register into which the current value of the memory location is loaded and `reg2` holds the value to be stored in the memory location by this atomic exchange (`reg2` could simply be the value 1 for a lock attempt, as in a test&set).

```
lock:   ll   reg1, location     /*load-locked the location to reg1*/
        bnz  reg1, lock         /*if location was locked (nonzero),
                                try again*/
        sc   location, reg2     /*store reg2 conditionally into location*/
        beqz  lock              /*if store-conditional failed, start again*/
        ret                     /*return control to caller of lock*/
```

and

```
unlock: st location, #0         /*write 0 to location*/
        ret                     /*return control to caller*/
```

Many processors may perform the LL at the same time, but only the first one that manages to put its store-conditional on the bus will actually succeed in its store-conditional. This processor will have succeeded in acquiring the lock, whereas the others will have failed and will have to retry the LL-SC. Note that the store-conditional may fail either because it detects the occurrence of an intervening write before even attempting to access the bus or because it attempts to get the bus but some other processor's store-conditional gets there first. Of course, if the location is 1 (nonzero) when a process does its LL, it will load 1 into `reg1` and will retry the lock starting from the LL without even attempting the store-conditional.

It is worth noting that the LL itself is not a lock and the store-conditional itself is not an unlock. For one thing, the completion of the LL itself does not imply obtaining exclusive access; in fact, LL and store-conditional are used together to implement a lock operation. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomically with respect to those instructions on other processors, so in fact these instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes to the synchronization variable itself intervene between the LL and store-conditional. In fact, since the instructions between the LL and store-conditional are executed unconditionally but should not be visible if the store-conditional fails, it is important that they do not modify any other important state. Typically, these instructions manipulate only the register into which the synchronization variable is loaded—for example, to perform the op part of a fetch&op—and do not modify any other program variables (modification of this register is okay since the register will be reloaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline and, in fact, often specify what instructions are possible to insert with a guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and store-conditional should also be kept small to reduce the probability of store-conditional failure due to an intervening write. Although the LL and store-conditional do not constitute a lock-unlock pair, they can be used directly to implement certain atomic operations on shared data structures. For example, if the desired function is a small operation on a globally shared variable (like a counter or global sum), it makes much more sense to implement it as the natural sequence (LL, register op, store-conditional, test) than to build a lock and unlock around the variable update.

Like the test-and-test&set, the spin-lock built with LL-SC does not generate bus traffic during the waiting algorithm if the LL indicates that the lock is currently held. Better than the test-and-test&set, it also does not generate invalidations on a failed attempt to obtain the lock (i.e., a failed store-conditional). However, when the lock is released, the processors spinning in a tight loop of load-locked operations will indeed miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition by the processor whose store-conditional succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case and there are no read-modify-write bus transactions, but traffic still increases linearly with the number of processors (i.e., $O(p)$ bus transactions per look acquisition). Since spinning on a locked location is already done through reads (load-locked operations), no analog of a test-and-test&set exists to further improve its performance. However, backoff can be used between the LL and store-conditional to reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and does not reduce traffic to a minimum. More advanced lock algorithms can be used that provide both fairness and reduced traffic. They can be built using either atomic read-modify-write instructions or atomic operations of equivalent semantics synthesized with LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms that are appropriate for bus-based machines.

Advanced Lock Algorithms

Especially when using an atomic exchange instruction like test&set, instead of LL-SC, to implement locks, it is desirable to have only one process actually attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the preceding algorithms). It is even more desirable to have only one process incur a read miss (even with LL-SC) when a lock is released. The *ticket lock* accomplishes the first purpose; the *array-based lock* accomplishes both goals but at a little cost in space. Unlike all the previous locks, both these locks are fair and grant the lock to processors in FIFO order.

Ticket Lock The ticket lock operates just like the ticket system in the sandwich line at a delicatessen or like the teller line at a bank. Every process wanting to acquire the lock takes a ticket number and then busy-waits on a global `now-serving` number—like the number on the LED display that we watch intently in the sandwich line—until the `now-serving` number equals the ticket number it obtained. To release the lock, a process simply increments the `now-serving` number so that the next waiting process can acquire the lock. The atomic primitive needed is a fetch&increment, which a process uses when it first reaches the lock operation to obtain its ticket number from a shared counter. No atomic operation (e.g., test&set) is needed to actually obtain the lock upon a release since only the unique process that has its ticket number equal to `now-serving` attempts to enter the critical section when it sees the release. Thus, the acquire method is the fetch&increment, the waiting algorithm is busy-waiting for `now-serving` to equal the ticket number, and the release method is to increment `now-serving`. This lock has uncontended latency about equal to the test-and-test&set lock but generates much less traffic. Although every process does a fetch&increment when it first arrives at the lock (presumably not every process at the same time), the test&set attempts upon a release of the lock are eliminated, which tend to be simultaneous and a lot more heavily contended. The ticket lock also requires constant and small storage and is fair since processes obtain the lock in the order of their fetch&increment operations.

The fetch&increment needed by the ticket lock can be implemented with LL-SC. However, since the simple LL-SC lock already avoids multiple processors issuing invalidations in trying to acquire a lock after its release, there is not a large difference in traffic between the ticket lock and the simple LL-SC lock. (The simple LL-SC lock is somewhat worse since in that case another invalidation and set of read misses occur when a processor succeeds in its store-conditional.) The key difference between these two locks is fairness.

Like the simple LL-SC lock, the ticket lock still has a read traffic problem at a release. The reason is that all processes spin on the same variable (`now-serving`). When that variable is written at a release, all processors' cached copies are invalidated, and they all incur a read miss. The read misses may be combined on some buses but can cause unnecessary traffic if the combining is unavailable or unsuccessful. One way to reduce this bursty read-miss traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so that none tries to acquire it for a while. A promising technique is to have each processor back off from trying to read the `now-serving` counter by a duration proportional to when it expects its turn to actually come—that is, by a duration proportional to the difference in its ticket number and the `now-serving` value it last read. Alternatively, the array-based lock completely eliminates this extra read traffic upon a release by having every process spin on a distinct location.

Array-Based Lock The idea here is to use a fetch&increment to obtain not a value but a unique location on which to busy-wait. If there are p processes that might possibly compete for a lock, then the lock data structure contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound), the waiting method spins on this location, and the release method writes a value denoting "unlocked" to the next location in the array (after the one that the releasing processor was itself spinning on). Only the processor that was spinning on that next location has its cache block invalidated at the release; its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. Its uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is potentially more scalable than the ticket lock since only one processor incurs the read miss. For the same reason, unlike the ticket lock, it does not need any form of backoff to reduce traffic. Its only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small, this is usually not a very significant drawback. It has a potential drawback for machines with distributed memory, but we shall discuss this drawback and lock algorithms that overcome it in Chapter 7.

Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5.30. All locks are implemented using LL-SC since the Challenge provides only these and not atomic instructions. Results are shown for a somewhat more parameterized version of the earlier microbenchmark code, in which a process is allowed to insert a delay not only for the critical section but also between its release of the lock and its next attempt to acquire it (as will happen in a real program). That is, the code is a loop over the following body:

```
lock(L);
critical_section(c);
unlock(L);
delay(d);
```

Let us consider three cases: (1) $c = 0$, $d = 0$; (2) $c = 3.64$ μs, $d = 0$; and (3) $c = 3.64$ μs, $d = 1.29$ μs—called the *null* critical section case, the *non-null* critical section case, and the non-null critical section with *delay* case, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases, the delays c and d (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure only the total time taken for a certain number of lock acquisitions and releases (see also Exercise 5.15).

Consider the null critical section case. The first observation, comparing Figure 5.30 with Figure 5.29, is that all the other locks are indeed better than the test&set locks, as expected.[9] The second observation is that the simple LL-SC locks actually seem to perform better than the more sophisticated ticket lock and array-based lock. For these locks, which don't encounter as much contention as the test&set lock, performance is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that releases a lock with a write follows it immediately with the read (LL) for its next acquire, its read and the subsequent store-conditional are likely to succeed in its cache before

9. The test&set is simulated using LL-SC as follows: every time a store-conditional fails, a write is performed to another variable in the same cache block, causing invalidations as a test&set would. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

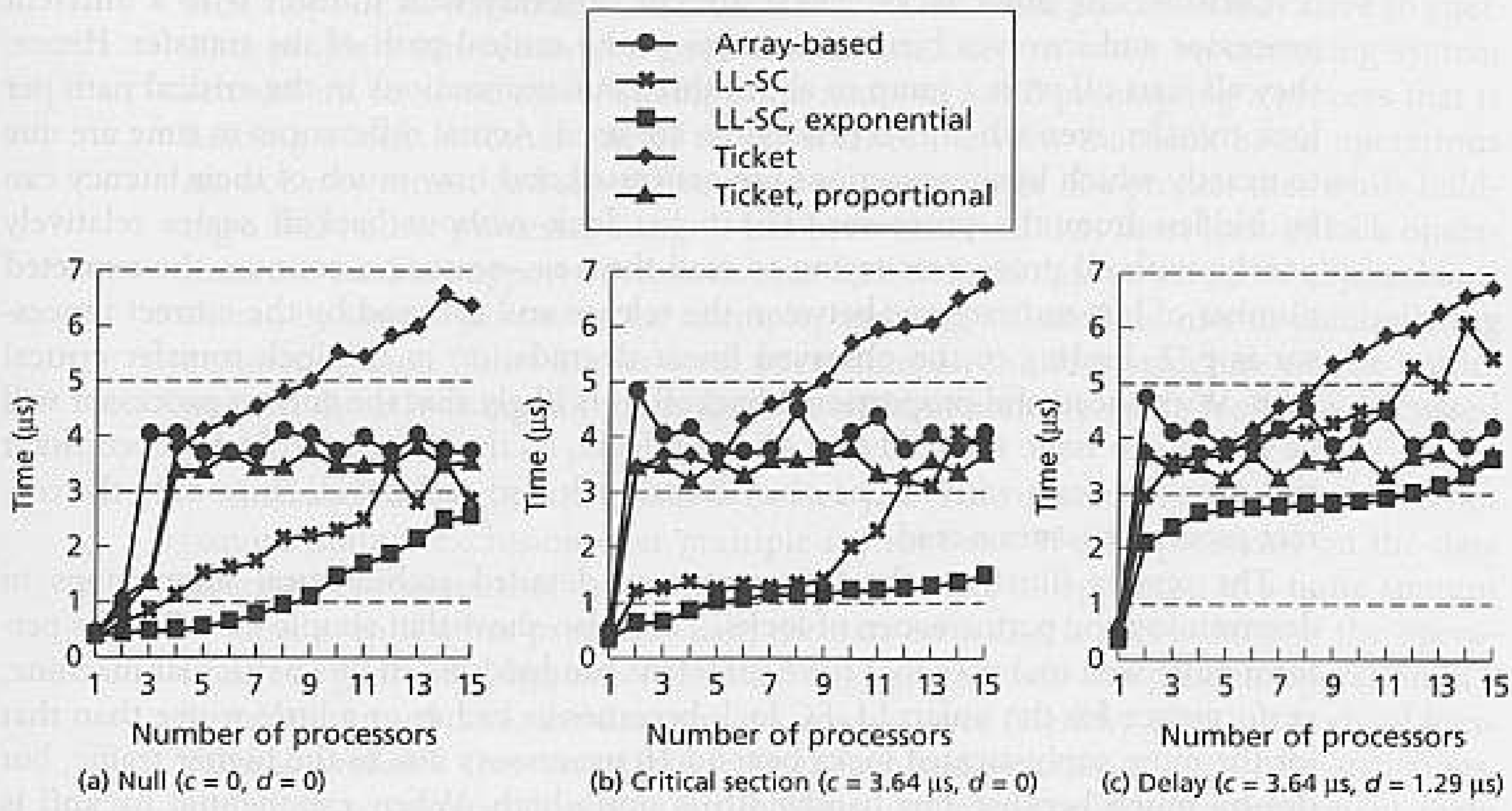

FIGURE 5.30 Performance of locks on the SGI Challenge for three different scenarios

another processor can read the block across the bus. (The bias on the SGI Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the read exclusive corresponding to the releasing write gets out on the bus.) Lock transfer is very quick, and performance is good, but the same processor keeps acquiring the lock repeatedly. As the number of processors and the competition for the bus increase, the likelihood of the last releaser's store-conditional successfully obtaining the bus decreases, and hence the likelihood of self-transfers decreases. In addition, bus traffic increases due to invalidations and read misses, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling, and a nonzero critical section ($c = 3.64$, $d = 0$) helps this along further.

With delays both inside and outside the critical section ($c = 3.64$, $d = 1.29$), we see the LL-SC lock not doing quite as well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even with two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is nonzero. This is because it is quite likely that while the processor that just released the lock is waiting for d to expire before doing its next acquire, all the other processors are in a backoff period and not even trying to acquire the lock. In the $d = 0$ case, the releasing processor reacquires the lock right away, especially with a small number of processors. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence, they all start off with a jump to about three bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to exactly which bus transactions are generated and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: with all processors trying to read the `now-serving` counter, the expected number of bus transactions between the release and the read by the correct processor is $p/2$, leading to the observed linear degradation in the lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer is constant and does not scale with p. The array-based lock also scales well since only the correct processor issues a read.

The results illustrate the importance of detailed architectural interactions in determining the performance of locks. They also show that simple LL-SC locks perform quite well on buses that have sufficient bandwidth. On this particular machine, performance for the unfair LL-SC lock becomes as bad as or a little worse than that for the more sophisticated locks beyond 16 processors due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. However, these results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparisons depend on what the synchronization patterns look like in practice—in real applications. For example, the effect of critical section and delay size on the frequency of self-transfers has a substantial impact on the comparison of unfair locks with fair locks. The nonrepresentativeness of the null case in this regard is therefore an important methodological consideration. An experiment to use LL-SC while guaranteeing round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts. Especially if fairness is desired, the ticket lock with proportional backoff and the array-based lock perform very well on bus-based machines.

Lock-Free, Nonblocking, and Wait-Free Synchronization

An additional set of performance concerns involving synchronization arises when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information about the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the parallel program as a whole slows down when one process is slowed. With traditional locks, the problem can be serious: if a process holding a lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers. In some cases, attempts are made to avoid preempting a process that is holding a lock. Another line of research takes the view that lock-based operations are not very robust and should be avoided; for example, if a process dies while holding a lock, other processes hang. It has been observed that most lock-unlock operations are used to support operations on a well-defined data structure or object that is shared by several processes, for example, updating a shared counter or manipulating a shared queue. These higher-level operations on the data structure can be implemented directly using atomic primitives without actually using locks, as discussed for LL-SC earlier.

A shared data structure is said to be *lock-free* if the operations defined on it do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in a finite amount of time, even if other processes halt, the data structure is *nonblocking*. If the operations can guarantee that every (nonfaulting) process will complete its operation in a finite amount of time, the data structure is *wait-free* (Herlihy 1993). A body of literature is available that investigates the theory and practice of such data structures, including requirements placed on the basic atomic primitives to implement them (Herlihy 1988), general-purpose techniques for translating sequential operations to nonblocking concurrent operations (Herlihy 1993), specific useful lock-free data structures (Valois 1995; Michael and Scott 1996), operating system implementations (Massalin and Pu 1991; Greenwald and Cheriton 1996), and proposals for architectural support (Herlihy and Moss 1993). The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made (reminiscent of LL-SC). As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy, and the result is put in a second register. Next, a compare&swap updates the shared counter only if its value is still the same as the copy. For more sophisticated, linked-list data structures, a new element is created and then linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve robustness, although it can be difficult to make them efficient.

Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it has been found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number of processors, whereas more sophisticated atomic operations like compare&swap and swapping the values of two memory locations are powerful enough to make this guarantee (Herlihy1988).

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point, and then barrier, event synchronization.

5.5.4 Point-to-Point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented by busy-waiting on ordinary variables, using them as flags. If we want to use blocking instead of busy-waiting, we can use semaphores, just as they are used in concurrent programming and operating systems (Tanenbaum and Woodhull 1997).

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable *a*, then a flag can be used to manage the synchronization as follows:

P_1	P_2
`a = f(x);` */*set a*/*	`while (flag is 0) do nothing;`
`flag = 1;`	`b = g(a);` */*use a*/*

If we know that the variable *a* is initialized to a certain value (say, 0), which will be changed to a new value we are interested in by this production event, then we can use *a* itself as the synchronization flag, as follows:

P_1	P_2
`a = f(x);` */*set a*/*	`while (a is 0) do nothing;`
	`b = g(a);` */*use a*/*

This eliminates the need for a separate flag variable and saves the write to and read of that variable at perhaps some cost in readability and maintainability.

Hardware Support: Full-Empty Bits

This idea of special flag values has been extended in some research machines (although mostly in machines with physically distributed memory) to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a *full-empty bit*, is associated with every word in memory. This bit is set when the word is "full" with newly produced data (i.e., on a write) and unset when the word is "emptied" by a processor consuming that data (i.e., on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location, it does so only if the full-empty bit is set to empty and then leaves the bit set to full. The consumer reads the location only if the bit is full and then sets it to empty. Hardware preserves the atomicity of the read or write with the manipulation of the full-empty bit. Given full-empty bits, our preceding example can be written without the spin loop as

P_1	P_2
`a = f(x);` */*set a*/*	`b = g(a);` */*use a*/*

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer-multiple-consumer synchronization or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter method requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers; see Chapter 2). For these reasons, and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine, there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs, special hardware was provided to monitor the priority of the process on each processor and to deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value, and most modern machines use simple arbitration strategies. In addition, a memory-mapped interrupt control region usually exists, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.5.5 Global (Barrier) Event Synchronization

Finally, let us examine barrier synchronization on a bus-based machine. Software algorithms for barriers are typically implemented using locks, shared counters, and flags. Let us begin with a simple barrier among p processes, which is called a *centralized barrier* since it uses only a single lock, a single counter, and a single flag.

Centralized Software Barrier

A shared counter maintains the number of processes that have arrived at the barrier and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, a process checks to see if the counter equals p, that is, if it is the last process to have arrived. If not, it busy-waits on the flag associated with the barrier; if so, it writes the flag to release the $p-1$ waiting processes. A simple attempt at a barrier algorithm may therefore look like

```
struct bar_type {
   int counter;
   struct lock_type lock;
   int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
   LOCK(bar_name.lock);
   if (bar_name.counter == 0)
      bar_name.flag = 0;                  /*reset flag if first to reach*/
   mycount = bar_name.counter++;          /*mycount is a private variable*/
   UNLOCK(bar_name.lock);
   if (mycount == p) {                    /*last to arrive*/
      bar_name.counter = 0;               /*reset counter for next barrier*/
      bar_name.flag = 1;                  /*release waiting processes*/
   }
   else
      while (bar_name.flag == 0) {};      /*busy-wait for release*/
}
```

Centralized Barrier with Sense Reversal

Can you see a problem with the preceding barrier? There is one. It occurs when the barrier operation is performed consecutively using the same barrier variable—for example, if each processor executes the following code:

```
some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);
```

The first process to enter the barrier the second time reinitializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see the flag change to 1 will exit the barrier, perform the subsequent computation, and enter the barrier again. However, suppose one processor P_x does not see the flag change from the first barrier before others have reentered the barrier for the second time; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. In the meantime, other processes may have already entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag can only get set to 1 again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop from the first barrier.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier until all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier and to not let a process reset the flag in a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. On the other hand, with the current setup we cannot wait for all processes to reach the barrier before resetting the flag to 0, since that is when we actually set the flag to 1 for the release. A better solution is to avoid explicitly resetting the flag value altogether and rather have processes wait for the flag to obtain a different release value in consecutive instances of the barrier. For example, processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time. Hence we call this method *sense reversal*. Now, in the previous example, the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed once when all processes have reached the (new) barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense reversal:

```
BARRIER (bar_name, p)
{
    local_sense = !(local_sense);           /*toggle private sense variable*/
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;           /*mycount is a private variable*/
    if (bar_name.counter == p) {            /*last to arrive*/
        UNLOCK(bar_name.lock);
        bar_name.counter = 0;               /*reset counter for next barrier*/
        bar_name.flag = local_sense;        /*release waiting processes*/
    }
    else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {};  /*busy-wait for
                                                    release*/
    }
}
```

Note that the lock is not released immediately after the increment of the counter but only after the condition is evaluated; the reason for this is revealed in an exercise (see Exercise 5.18). We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next.

(Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.)

Performance

The major performance goals for a barrier are similar to those for locks. They include the following:

- *Low latency (small critical path length).* The chain of dependent operations and bus transactions needed for p processors to pass the barrier should be small.
- *Low traffic.* Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. The barrier algorithm should reduce the total number of bus transactions (whether in the critical path or not) and hence the possible contention.
- *Scalability.* Latency and traffic should increase slowly with the number of processors.
- *Low storage cost.* We would, of course, like to keep the storage cost low.
- *Fairness.* We should ensure that the same processor does not always become the last one to exit the barrier (or we may want to preserve FIFO ordering).

In the centralized barrier described previously, each processor accesses the lock once, hence the critical path length is at least proportional to p. Consider the bus traffic. To complete its operation, a centralized barrier involving p processors performs $2p$ bus transactions for processors to obtain the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another $p - 1$ bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for even a test-and-test&set lock to be acquired by p processes because, in that case, each of the p releases causes an invalidation that results in $O(p)$ processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, so barriers can be expensive.

Improving Barrier Algorithms for a Bus

One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause fewer processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Section 3.3.2). In a binary combining tree, for example, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors ever access a given variable. In a distributed network with multiple parallel paths, such as those found in scalable machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other

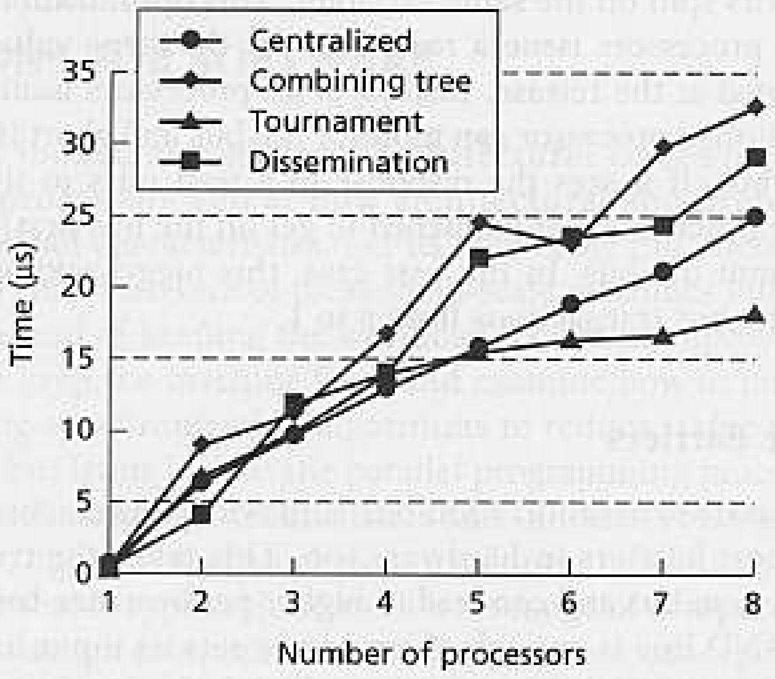

FIGURE 5.31 Performance of some barriers on the SGI Challenge. Performance is measured as average time per barrier over a loop of many consecutive barriers (with no work or delays between them). The higher critical path latency of the combining tree barrier hurts it on a bus, where it has no traffic and contention advantages.

in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables, they all generate bus transactions and hence serialization and contention on the same bus. Since a binary tree with p leaves has approximately $2p$ nodes, a combining tree requires a similar total number of bus transactions to the centralized barrier. It also has higher latency since, while it too requires $O(p)$ serialized bus transactions in all, even without bus serialization each processor must wait at least $\log p$ steps to get from the leaves to the root of the tree, each with significant work. The advantage of a combining tree for a bus is that it does not use locks but, rather, simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5.31. Some of the other barriers shown in the figure for illustration will be discussed along with tree barriers in the context of scalable machines in Chapter 7.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads and writes, the hardware primitives needed depend on which lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions for read misses in the centralized barrier (as well as for highly contended locks in which processors spin on the same variable). This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read-miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus, if it sees the response to a read miss to the same location (issued by another processor that happened to get on the bus first), and simply take the return value from the bus. In the best case, this piggybacking can reduce the number of read-miss bus transactions from p to 1.

Hardware Barriers

If a separate synchronization bus is provided, as discussed for locks, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus and can lead to higher-performance barriers. Conceptually, a single wired-AND line is enough. A processor sets its input high when it reaches the barrier and waits until the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers can be particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and that need global synchronization after every innermost loop. However, its value in practice is unclear, and it can be difficult to manage when only a portion of the processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier or to adapt the configuration of participating processors when processes are migrated among processors by the operating system. Having multiple participating processes running on the same processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support but build barriers in software out of locks and shared variables.

5.5.6 Synchronization Summary

Some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers. However, concerns about flexibility have led most contemporary designers to provide support for only simple atomic operations in hardware and to synthesize higher-level synchronization operations from them in software libraries. The application programmer generally uses the libraries and can be unaware of the low-level atomic operations supported on the machine. The atomic operations may be implemented either as single instructions or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters.