Efficient Extensible Synchronization in Sather

Jürgen W. Quittek, Boris Weissman

International Computer Science Institute, 1947 Center St., Berkeley, CA 94704
quittek@icsi.berkeley.edu, borisv@icsi.berkeley.edu,

Abstract. Sather, a parallel object-oriented programming language developed at ICSI, offers advanced thread synchronization constructs separating locking mechanism and policies. While a lock management system provides a general locking mechanism, synchronization objects define and implement different extensible policies. Commonly used synchronization objects such as mutual exclusion and reader/writer locks are provided by the standard Sather library. Synchronization objects with more complex semantics can be defined by the user. The conjunctive and disjunctive acquisition of collections of locks and the deadlock detection are distinct features of Sather supported by the locking mechanism.

This paper introduces the Sather synchronization constructs and presents the design and implementation of a lock management runtime system. We argue that a clean, object-oriented design allows us to support sophisticated synchronization policies while preserving efficiency on distributed computing platforms. The system is fully implemented and runs on several platforms including a network of symmetric multiprocessors connected by a fast, user-level, low latency communications network.

1 Introduction

Simple synchronization objects such as mutual exclusion locks or semaphores are under explicit user control in many parallel systems. However, in distributed or fault-tolerant environments runtime synchronization systems may be necessary to deal with added complexity. For example, distributed lock managers are used for access control in recoverable distributed file servers such as VAXcluster [1] or the Highly Available Network File Server [2]. Transactions in complex object-oriented DBMSs [3] and in distributed databases such as the Oracle Parallel Server [4] are also controlled by lock managers. Lock managers are also used to ensure memory consistency in distributed shared memory systems [5, 6].

The Sather runtime is more general than the examples listed above because it supports arbitrary locking policies. File systems use a mutual exclusion locking policy. Databases and distributed shared memory systems can add reader/writer locking policies. In Sather, synchronization objects define their own policies. Consequently, the common lock management runtime mechanism must be independent of the policies, but general enough to realize any policy.

The Arjuna programming System, a fault-tolerant distributed extension of C++ is an example of a system that provides separation between the synchronization mechanism and policies [7]. Another example is PRESTO, a parallel object-oriented environment for C++ [8]. However, in both Arjuna and PRESTO, only single locks can be acquired, while Sather supports atomic locking of sets of locks (conjunctive locking) and alternative locking of sets of locks (disjunctive locking). This functionality impacts overall design decisions for distributed environments.

2 Thread Synchronization in Sather

Sather is a modern parallel object-oriented programming language designed to be safe and efficient. Some major Sather features include strong static typing, dynamic dispatch, multiple inheritance, parametrized classes, iteration abstraction, garbage collection, and exception handling. Sather is available on most Unix platforms as well as on Windows NT. A more detailed language description can be found in [9, 10]. The Sather language specification, language manual, and full Sather distribution are available at http://www.icsi.berkeley.edu/~sather.

Sather supports concurrent programming based on threads. It offers a shared memory model for distributed platforms and allows the explicit placement of threads and objects.

Runtime synchronization is handled by a *lock management system (LMS)* and by *locks*. The LMS provides an advanced locking mechanism that avoids typical deadlocks and starvation. Locks define locking policies such as a mutual exclusion or reader/writer policy. While it is impossible to foresee all synchronization policies that may be required by different applications, Sather provides a library of commonly used ones. Applications that use more sophisticated synchronization patterns can define their own synchronization policies and extend the standard library. The functionality of the LMS itself is well-defined and supported by the Sather runtime.

2.1 The Lock Statement

The Sather lock statement provides a programming language level interface to the LMS. Simple applications of the Sather lock statement include critical sections [11] and are similar to the synchronized block construct in Java [12]. The lock statement also extends and generalizes many other common synchronization concepts.

A lock statement consists of one or more lists of Sather statements called branches. Each branch is associated with a set of locks. The following code example shows different usage patterns for the lock statement including conjunctive and disjunctive locking.

```
-- declare lock variables
lock1,lock2,lock3:$LOCK;
...
-- single lock synchronization
lock when lock1 then <branch 1> end;
...
-- acquire all three locks atomically
lock when lock1,lock2,lock3 then <branch 2> end;
...
```

```
-- Take either branch 3 if lock1 is available or branch 4
-- if lock2 and lock3 are available. Otherwise, take branch 5
lock
   when lock1 then <branch 3>
   when lock2,lock3 then <branch 4>
   else <branch 5>
end;
```

Locks are synchronization objects which can be acquired, held and released by threads. A branch of a lock statement can be taken only if all locks associated with this branch have been acquired. Only a single branch is guaranteed to be executed, even if several branches can be taken. The *else branch* is executed if no other branch can be taken. In the absence of the else branch, the executing thread is blocked until at least one branch becomes available. After a branch has been executed, all held locks are released.

Acquiring a set of locks is an atomic operation. This allows to avoid many typical deadlock situations. The lock statement guarantees fairness with respect to threads and branches. A thread that can acquire a branch will eventually do so, even if this requires blocking other threads. Another aspect of fairness deals with selecting a branch when more than one branch are available. In this case, each branch is taken with a non-zero probability. This avoids starvation of individual branches.

2.2 Synchronization Objects: Locks

While the semantics of the lock statement is well-defined, the locks used in the statement may behave differently. The Sather distribution provides a library of locks including mutual exclusion locks, reader/writer locks and other more complex locks. Locks with other semantics can be defined by the user.

Any lock has to be a subtype of the abstract class `$LOCK`. In Sather, abstract classes define interfaces and provide no implementation. Any concrete lock class that inherits from `$LOCK` must fully implement its interface:

```
abstract class $LOCK is
   acquirable(tid:THREAD_ID):BOOL;
   acquire(tid:THREAD_ID);
   release(tid:THREAD_ID);
   request(tid:THREAD_ID);
   cancel_request(tid:THREAD_ID);
   wait_for(tid:THREAD_ID):ARRAY{THREAD_ID};
end;
```

The lock interface methods may be called only by the LMS. They cannot block, raise exceptions or recursively execute other lock statements.

`acquirable` returns `true` if the thread identified by `tid` can acquire this lock. Methods `acquire` and `release` notify the lock that it is has been acquired or released respectively. `request` is executed for each thread blocking on the lock. Method `cancel_request` is called, when a thread unblocks and can continues execution.

`wait_for` is used for deadlock detection. It returns an array of threads that have to release the lock before it can be acquired by the thread identified by `tid`.

The programmer's freedom to define the semantics of synchronization objects enables the customization of synchronization policies. For instance, Hoare's monitors [13] can be easily implemented and used in the Sather lock statement although they are not directly supported by the programming language.

Java is similar to Sather in many respects, but the Sather lock statement is more general and more powerful than Java's synchronized blocks and methods. A detailed comparison of the Sather lock statement with synchronization constructs of other languages can be found in [14].

3 The LMS Architecture

The major design decisions for the LMS involve selecting the representation (either active or passive) and choosing granularity.

3.1 Active versus Passive

A lock management system can be either active or passive. Functions of an active LMS are performed by one or more designated threads. A passive LMS consists of a set of (passive) objects which are accessed by threads requesting locking operations. A passive LMS might satisfy many non-conflicting requests concurrently. However, synchronization is necessary to serialize critical sections of conflicting requests. For an active LMS, less coordination is necessary since synchronization objects may be permanently assigned to managing threads performing LMS functions. An active LMS approach may lead to a simpler overall design, but has negative performance implications. The disadvantage of an active LMS is that for each lock statement two context switches are necessary (from the requesting thread to a an LMS thread and then back to the requesting thread) which is generally more expensive than providing safety for passive lock management.

The Sather runtime presented in this paper uses a passive LMS. A passive LMS was chosen because thread context switch time on the supported platforms is much longer than the time required to synchronize threads by low level synchronization primitives.

3.2 Granularity

Sather is a parallel programming languages designed to run on a network of symmetric multiprocessors, called clusters. This adds another dimension to the synchronization problem, namely granularity. Levels of granularity for a distributed LMS can be: (a) one instance per system; (b) one instance per cluster; (c) one instance per lock or group of locks; (d) one instance per lock per cluster. An earlier implementation of the LMS used a single instance per system [15]. But since even a single instance per cluster (b) leads to serialization for threads that otherwise would run completely asynchronously, an LMS of fine granularity is a better choice for fine-grain parallel systems.

The finest granularity distributed synchronization algorithms (d) for mutual exclusion were extensively investigated in earlier studies (see overview in [17]). However, since in the general case lock statements deal with local synchronization objects, an LMS with slightly lower granularity (c) was chosen. If needed, the finest granularity can be still achieved on the synchronization policy level.

3.3 Internal Structure

The details of the LMS design are best characterized by the interfaces of its subsystems. In addition to the lock interface (`$LOCK`), two more interfaces are introduced: a *lock manager interface* (`$LOCK_MANAGER`) and a *management access* interface (`$MANAGEMENT_ACCESS`). An object implementing the lock manager interface is uniquely assigned to each lock. It is the only object that may use the lock interface and call methods on the synchronization object. A lock manager may manage more than one lock and a lock manager for a particular synchronization object may change dynamically at runtime. In order to ensure thread safety and atomicity of locking operations, threads executing in the lock manager object must be protected by mutual exclusion.

When a lock is created, it is assigned a new lock manager. When the lock is used in a statement with more than one lock for the first time, a new manager is created. This new lock manager takes over all locks used in the statement.

The lock manager interface contains methods for acquiring, trying and releasing locks:

```
abstract class $LOCK_MANAGER is
   acquire(locks:ARRAY2{$LOCK}):INT;
   release(locks:ARRAY2{$LOCK});
   ...
end;
```

A two-dimensional array of locks is passed to these methods containing a row for each branch of the Sather lock statement. An empty row is used for an else branch. If there exists no such row that all its locks could be acquired, the executing thread is blocked until this condition changes. The condition is re-evaluated on each call to `acquire` or `release`. Since all changes of the synchronization object state are made known to the associated lock manager, this design is free of race conditions. The actual implementation of thread blocking uses the blocking mutual exclusion primitive available on a particular hardware platform.

The lock manager interface provides other methods for safe transition between lock managers when synchronization objects are used in combinations. The way a lock manager keeps sleeping threads is not specified because different locking policies might require different representations. However, the lock manager transition interface is strictly enforced.

Since lock managers are generated dynamically, the compiler cannot place them in static memory. Consequently, the locks themselves have to serve as entry points for lock management at runtime. For this purpose, all locks have to implement another in-

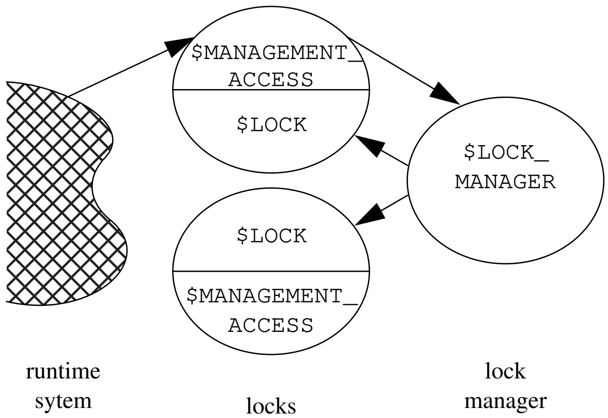

Figure 1: Access graph of the lock management

terface, the management access interface. It provides a safe way for finding a current manager for a given synchronization object.

When entering a lock statement, the runtime system accesses the appropriate lock manager via one of the statement's locks (Figure 1).

This design makes both simple and common cases fast. The standard Sather library locks implement not only `the $LOCK` and `$MANAGEMENT_ACCESS` interfaces, but also the `$LOCK_MANAGER` interface. Unless they are used in combinations in complex lock statements, simple locks are managed by specialized lock managers. This saves indirections and dynamic dispatches necessary in the general, but less common case. When locks are used in a more complex context, a more general lock manager is created and takes over the locks.

4 Implementation

The Sather parallel runtime system is highly portable. It includes its own user-level thread package, called Active Threads. Active Threads are available for Sparc V8 and V9, Intel 386 and higher, DEC Alpha AXP, and HPPA 1.1. Alternatively, proprietary thread systems, such as Solaris Threads and POSIX threads [16] may be used as a compilation target. Tables 1 and 2 show execution times of some basic Sather synchronization operations. For comparison, the times for the blocking mutual exclusion operations of the underlying synchronization primitives are also presented.

The numbers in Table 1 were obtained on a Sparc Ultra Enterprise 5000 server with 8 processors. The overhead for a Sather lock is relatively small, in some cases Sather locks are even faster than the basic primitives. The reason for this is that Sather lock managers are protected by spinlocks [18] which are much faster than blocking mutex primitives. If all locks can be acquired, the lock manager does not need to use mutex primitives; these are only required, if a thread has to be blocked. Since acquiring locks

Table 1: Lock access time in µs on an 8cpu Sun Enterprise 5000 server.

	Solaris Threads		Active Threads	
	basic mutex	Sather mutex	basic mutex	Sather mutex
acquire and release single mutex	0.73	0.99	0.83	0.66
try and fail single mutex	0.29	0.82	0.27	0.49
acquire and release two conjunctive locks	—	1.43	—	0.76

Table 2: Lock access time in µs on a network of 4cpu SPARCstation-10 SMPs connected by Myrinet

	Solaris Threads		Active Threads	
	basic mutex	Sather mutex	basic mutex	Sather mutex
acquire and release single local mutex	1.5	2.13	2.0	1.33
try and fail single local mutex	0.6	1.90	0.50	1.03
acquire and release two conjunctive locks locally	—	3.29	—	1.68
acquire and release single remote lock	—	309	—	221
try and fail single remote lock	—	158	—	112

in combinations is supported only by Sather, the corresponding cells for underlying thread systems are left blank.

Table 2 shows the same measurements on a distributed platform of SPARCstation-10 symmetric multiprocessors connected via Myrinet, a low latency packet switched network. Numbers for access to remote Sather locks are also provided.

5 Conclusion

The Sather lock management system displays a clean, object-oriented design that supports sophisticated synchronization policies while preserving high efficiency. The system is fully implemented and runs on several parallel and distributed platforms.

References

1. W.E. Snaman Jr., D.W. Thiel: The VAX/VMS Distributed Lock Manager. Digital Technical Journal, September 1987.
2. A. Bhide, S. Shepler: A highly available lock manager for HA-NFS. Proc. Summer 1992 Usenix Conf., Usenix Association, Berkeley, pp. 177-184, 1992.
3. L. Daynes, O. Gruber, P. Valduriez: Locking in OODBMS clients supporting nested transactions. Proc. 11th Intl. Conf. on Data Engineering, IEEE Computer Society Press, Los Alamitos, pp. 316-323, 1995.
4. M. Aldred, I. Gertner, S. McKellar: A Distributed Lock Manager on fault-tolerant MPP. Proc. 28th Hawaii Intl. Conf. on System Sciences, Vol. 1, IEEE Computer Society Press, Los Alamitos, pp. 134-136, 1995.
5. T. Stiemerling, T. Wilkinson, A. Saulsbury: Implementing DVSM on the TOPSY Multicomputer. Symposium on Experiences with Distributed and Multiprocessor Systems, Newport Beach, 1992, Usenix Assoc., Berkeley, pp. 263-278, 1992.
6. G. Hermannsson, L. Wittie: Fast Locks in Distributed Shared Memory Systems. Proceedings of the 27th Hawaii Intl. Conf. on System Sciences, Vol. 1, IEEE Comp. Soc. Press, Los Alamitos, pp. 574-583, 1994.
7. G.D. Parrington, S.K. Shrivastava, S.M. Wheater: The design and implementation of Arjuna. Computing Systems **8**(3):255-308, 1995.
8. B.N. Bershad, E.D. Lazovska, and H.M. Henry: Presto: A System for Object-Oriented Parallel Programming. Software-Practice and Experience **18**(8):713-732,1988.
9. S. Omohundro: The sather programming language. Dr. Dobb's Journal **18**(11):42-48, 1993.
10. C. Szyperski. S. Omohundro, S. Murer: Engineering a programming language: The type and class system of Sather. J. Gutknecht (Ed.), Programming Languages and System Architectures, Springer-Verlag, pp. 208-227, 1993.
11. C.A.R. Hoare: Towards a theory of parallel programming. Operating Systems Techniques, C.A.R. Hoare, R.H. Perrot (Eds.), Academic Press, London, pp. 61-71, 1972.
12. J. Gosling, B. Joy, G.L. Steele: The Java Language Specification. Addison-Wesley, Reading, Mass., 1996.
13. C.A.R. Hoare: Monitors: an operating system structuring concept. Comm. of the ACM **17**(11):549-557, 1974.
14. C.-C. Lim: A Parallel, Object-Oriented System for Realizing Reusable and Efficient Data Abstractions. Ph.D. thesis, University of California at Berkeley, 1993.
15. C. Fleiner: Parallel Optimizations: Advanced Constructs and Compiler Optimizations for a Parallel, Object-Oriented, Shared Memory Language Running on a Distributed System. Ph.D. thesis, University of Fribourg, Switzerland, 1997.
16. The Institute of Electrical and Electronics Engineers. Portable Operating System Interface (POSIX) - Part 1: Amendment 2: Threads Extensions [C Language]. POSIX P1003.4a/D7. April, 1993.
17. T. Johnson: A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms. Proceedings of the 9th International Parallel Processing Symposium, IEEE Computer Society Press, Los Alamitos, California, 1995.
18. T.E. Anderson: The performance of spin lock alternatives for shared multiprocessors. IEEE Transaction on Parallel and Distributed Systems **1**(1):6-16, January 1990.