(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 783 150 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
26.02.2003 Bulletin 2003/09

(51) Int Cl.[7]: **G06F 9/44**

(21) Application number: **96308764.8**

(22) Date of filing: **04.12.1996**

(54) **System, method, storage medium and computer-readable modules for space efficient object locking**

Vorrichtung, Verfahren, Speichermedium und computerlesbare Module zur raumeffizienten Objektverriegelung

Dispositif, procédé, support d'enregistrement et modules lisibles par ordinateur pour le verrouillage efficace en espace des objets

(84) Designated Contracting States:
DE FR GB IT NL SE

(30) Priority: **08.12.1995 US 569753**
30.04.1996 US 640244

(43) Date of publication of application:
09.07.1997 Bulletin 1997/28

(73) Proprietor: **SUN MICROSYSTEMS, INC.**
Mountain View, CA 94043 (US)

(72) Inventors:
- **Joy, William N.**
  Aspen, Colorado 81612 (US)
- **Van Hoff, Arthur A.**
  Mountain View, California 94043 (US)

(74) Representative:
Cross, Rupert Edward Blount et al
BOULT WADE TENNANT,
Verulam Gardens
70 Gray's Inn Road
London WC1X 8BT (GB)

(56) References cited:
- **MOHAN C ET AL: "Efficient locking and caching of data in the multisystem shared disks transaction environment" ADVANCES IN DATABASE TECHNOLOGY - EDBT '92. 3RD INTERNATIONAL CONFERENCE ON EXTENDING DATABASE TECHNOLOGY PROCEEDINGS, VIENNA, AUSTRIA, 23-27 MARCH 1992, ISBN 3-540-55270-7, 1992, BERLIN, GERMANY, SPRINGER-VERLAG, GERMANY, pages 453-468, XP002055779**
- **PRASAD VISHNUBHOTLA: "SYNCHRONIZATION AND SCHEDULING IN ALPS OBJECTS" INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, SAN JOSE, JUNE 13 - 17, 1988, no. 1988, 13 June 1988, INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, pages 256-264, XP000042006**
- **SANTOSH K SHRIVASTAVA ET AL: "STRUCTURING FAULT-TOLERANT OBJECT SYSTEMS FOR MODULARITY IN A DISTRIBUTED ENVIRONMENT" IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, vol. 5, no. 4, 1 April 1994, pages 421-432, XP000439637**

EP 0 783 150 B1

Printed by Jouve, 75001 PARIS (FR)

Description

[0001] The present invention relates generally to object-oriented computer systems in which two or more threads of execution can be synchronized with respect to an object, and particularly to a system, method, storage medium and computer-readable modules for efficiently allocating lock data structures in a system where most or all objects are lockable, but relative few objects are in fact ever locked.

BACKGROUND OF THE INVENTION

[0002] In multiprocessor computer systems, software programs may be executed by threads that are run in parallel on the processors that form the multiprocessor computer system. As a result, a program may be run in parallel on different processors since concurrently running threads may be executing the program simultaneously. Moreover, if a program can be broken down into constituent processes, such computer systems can run the program very quickly since concurrently running threads may execute in parallel the constituent processes. Single processor, multitasking computer systems can also execute multiple threads of execution virtually simultaneously through the use of various resource scheduling mechanisms well known to those skilled in the art of multitasking operating system design.

[0003] The programs run on such computer systems are often object oriented. In other words, the threads executing the programs may invoke methods of objects to perform particular functions. However, some methods of some objects may be implemented only one at a time because of hardware or software constraints in the computer system. For example, an object may require access to a shared computer resource, such as an I/O device, that can only handle one access by one thread at a time. Thus, since concurrently running threads may concurrently seek to invoke such an object, the object must be synchronized with only one thread at a time so that only that thread has exclusive use to the object (i.e., only one the thread at a time can own a lock on the object).

[0004] In the past, various approaches have been used to synchronize an object with a thread. These include the use of synchronization constructs like mutexes, condition variables, and monitors. For instance, P. Vishnubhotla: "Synchronization and Scheduling in ALPS Objects", International Conference on Distributed Computing Systems, San Jose, June 1988 uses for each object a high priority process called manager to implement the synchronization and S.K. Shrivastava et al.: "Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment", IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 4, April 1994 associates to each object a lock manager object. When using monitors, each monitor identifies the thread that currently owns the object and any threads that are waiting to own the object. However, in the computer systems that employ these monitors there is often a monitor for every synchronizable object. As a result, this approach has the distinct disadvantage of requiring a large amount of memory.

[0005] A simple approach to reducing the memory required by these monitors is to allocate a cache of monitors and to perform a hash table lookup in this cache on each monitor operation. Such an approach can substantially increase the overhead of monitor operations, and the slow speed of this solution led to the present invention.

[0006] Embodiments of the present invention provide a object locking system in which space is allocated for lock data on an as-needed basis so as to avoid the allocation of memory resources for lock data structures for objects that, while lockable, are in fact never locked.

[0007] Embodiments of the present invention also provide a lock data allocation system and method that is computationally efficient and that imposes essentially no computational overhead for frequently used object, and that uses storage resources that are proportional to the number of locked objects.

SUMMARY OF THE INVENTION

[0008] In summary, the present invention is a multithreaded computer system having a memory that stores a plurality of objects and a plurality of procedures. Each object has a lock status of locked or unlocked. There are two ways objects may be represented, either as a handle consisting of a data pointer to a data structure and methods pointer to a methods array, or as a direct pointer to an object data structure, the first element of which is a methods pointer to a methods array. For the purposes of the present invention, this representational difference is not significant.

[0009] In either case, the methods array includes lock and unlock procedures for the object. Each object is further an instance of some class, and has a data reference, stored with its method array, to a class data structure associated with this class. Two objects that are instances of the same class then share this class data structure, having identical references to it. This class data structure includes a permanently allocated class lock data structure associated with this class, which can be used for synchronization of modifications to objects which otherwise have no lock data structures associated with them.

[0010] The system uses a single global object locking procedure as the lock procedure to service lock requests on objects that have not been allocated a lock data subarray (i.e., objects that have never been locked and objects that have not recently been locked), and uses a local, object-specific locking procedure as the lock procedure to service

lock requests on objects that have been allocated a lock data subarray (i.e., objects that are locked and objects that have been recently locked).

[0011] The global object locking procedure has instructions for creating a local object locking procedure specifically for the object to be locked. The local object-specific locking procedure includes as private data a lock data subarray for storing lock data. The local object-specific locking procedure has instructions for updating that object's stored lock data. A lock data cleanup procedure, executed when the system's garbage collection procedure is executed, releases the memory used for the local object-specific locking procedure if the object has not been recently locked.

[0012] In a preferred embodiment, each object that has not been allocated a lock data subarray has a methods pointer that references a set of procedures that includes the global object locking procedure; such objects are necessarily never in a locked condition. Each object that has been allocated a local object-specific locking procedure has a methods pointer that references a set of procedures that includes its local object-specific locking procedure. Furthermore, the global object locking procedure includes instructions for updating a specified object's method pointer to point to a set of procedures that includes the local object-specific locking procedure.

[0013] The lock data cleanup procedure includes instructions, activated when a specified object's updated lock data indicate that the specified object has not been recently locked, for changing the speccified object's method pointer to point to a set of procedures that includes the global object locking procedure.

[0014] More specifically, in a preferred embodiment the computer system includes a set of object classes, and each object class includes a virtual function table (VFT) that includes pointers referencing a set of methods associated with the object class as well as a pointer that references the global object locking procedure. Each object that has not been recently locked has a methods pointer that references the VFT for a corresponding object class.

[0015] For each object that has been recently locked, the system stores an object-specific virtual function table (VFT) that includes pointers referencing the set of methods associated with its object class as well as a pointer that references that object's local object locking procedure. The global object locking procedure includes instructions for creating the object-specific VFT and for updating a specified object's method pointer to reference its object-specific VFT.

BRIEF DESCRIPTION OF THE DRAWINGS

[0016] Examples of the invention will be described in conjunction with the drawings, in which:

Fig. 1 is a block diagram of a computer system incorporating a preferred embodiment of the present invention.

Fig. 2 is a block diagram of the data structure for an object that has not yet been allocated a lock data subarray in a preferred embodiment of the present invention.

Fig. 3 is a block diagram of the data structure for an object for which a lock data subarray has been allocated in a preferred embodiment of the present invention.

Figs. 4 and 5 are flow charts of the procedures for locking an object in a preferred embodiment of the present invention.

Fig. 6 is a flow chart of a preferred embodiment of a lock data cleanup method.

Fig. 7 is a flow chart of an alternate embodiment of a lock data cleanup method.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0017] Referring to Fig. 1, there is shown a distributed computer system 100 having multiple client computers 102 and multiple server computers 104. In the preferred embodiment, each client computer 102 is connected to the servers 104 via the Internet 103, although other types of communication connections could be used. While most client computers are desktop computers, such as Sun workstations, IBM compatible computers and Macintosh computers, virtually any type of computer can be a client computer. In the preferred embodiment, each client computer includes a CPU 105, a communications interface 106, a user interface 107, and memory 108. Memory 108 stores:

- an operating system 109;
- an Internet communications manager program 110;
- a bytecode program verifier 112 for verifying whether or not a specified program satisfies certain predefined integrity criteria;
- a bytecode program interpreter 114 for executing application programs;

- a class loader 116, which loads object classes into a user's address space and utilizes the bytecode program verifier to verify the integrity of the methods associated with each loaded object class;
- at least one class repository 120, for locally storing object classes 122 in use and/or available for use by users of the computer 102;
- at least one object repository 124 for storing objects 126, which are instances of objects of the object classes stored in the object repository 120.

[0018] In the preferred embodiment the operating system 109 is an object-oriented multitasking operating system that supports multiple threads of execution within each defined address space. The operating system furthermore uses a garbage collection procedure to recover the storage associated with released data structures. The garbage collection procedure is automatically executed on a periodic basis, and is also automatically invoked at additional times when the amount of memory available for allocation falls below a threshold level. For the purposes of this document it may be assumed that all objects in the system 109 are lockable objects, although in practice relatively few objects are actually ever locked.

[0019] The class loader 116 is typically invoked when a user first initiates execution of a procedure, requiring that an object of the appropriate object class be generated. The class loader 116 loads in the appropriate object class and calls the bytecode program verifier 112 to verify the integrity of all the bytecode programs in the loaded object class. If all the methods are successfully verified an object instance of the object class is generated, and the bytecode interpreter 114 is invoked to execute the user requested procedure, which is typically called a method. If the procedure requested by the user is not a bytecode program and if execution of the non-bytecode program is allowed (which is outside the scope of the present document), the program is executed by a compiled program executer (not shown).

[0020] The class loader is also invoked whenever an executing bytecode program encounters a call to an object method for an object class that has not yet been loaded into the user's address space. Once again the class loader 116 loads in the appropriate object class and calls the bytecode program verifier 112 to verify the integrity of all the bytecode programs in the loaded object class. In many situations the object class will be loaded from a remotely located computer, such as one of the servers 104 shown in Fig. 1. If all the methods in the loaded object class are successfully verified, an object instance of the object class is generated, and the bytecode interpreter 114 is invoked to execute the called object method.

[0021] Synchronized methods are defined for the purposes of this document to be methods that include using a locking methodology so as to limit the number of threads of execution (hereinafter "threads") that can simultaneously use a system resource. The most common synchronization tool is a mutex, which enables only a single thread to use a particular system resource at any one time, and which includes a mechanism for keeping track of threads of execution waiting to use the resource. While the synchronization mechanism described in this document is a mutex type of locking mechanism, the methodology of the present invention is equally applicable to computers system having other synchronization mechanisms, including but not limited to semaphores, time based lock expirations, and so on.

[0022] In the context of the preferred embodiment, a synchronized method is always synchronized on a specific object. For example, multiple threads may execute synchronized methods that call for exclusive use of a system resource represented by a specific object. When any one of the threads has "possession" of the lock on the object, all other threads that request possession of the lock on the object are forced to wait until all the earlier threads get and then release the lock on the object.

Data Structures for Unlocked and Locked Objects

[0023] Fig. 2 shows the data structure 200 in a preferred embodiment of the present invention for an object that has not been recently locked. As will be described next, all such objects are, necessarily, unlocked, and furthermore have not been allocated a lock data subarray. In one preferred embodiment, the phrase "object X has not been recently locked" is defined to mean that object X has not been locked since the last garbage collection cycle by the operating system. In other preferred embodiments, the term "recently" may be defined as a predefined amount of time, such as a certain number of seconds, or as the period of time since a dependable periodic event in the computer system other than the execution of the garbage collection procedure.

[0024] An object of object class A has an object handle 202 that includes a pointer 204 to the methods for the object and a pointer 206 to a data array 208 for the object.

[0025] The pointer 204 to the object's methods is actually an indirect pointer to the methods of the associated object class. More particularly, the method pointer 204 points to the Virtual Function Table (VFT) 210 for the object's object class. Each object class has a VFT 210 that includes: (A) pointers 212 to each of the methods 214 of the object class; (B) a pointer 215 to a global lock method (Global Lock1) 216 for synchronizing an object to a thread; (C) a pointer 217 to a special Class Object 218; and (D) a pointer 220 to an unlock method 221 for releasing the lock monitors. There is one Class Object 218 for each defined object class, and the Class Object includes a permanently allocated lock data

subarray (sometimes called a lock monitor) 219 for the class. The Class Object 218 is used in the preferred embodiment to synchronize access to the lock data subarrays of all objects that are instances of the corresponding object class.

[0026] As shown in Fig. 2, there is only one copy of the VFT 210 and the object methods 214 for the entire object class A, regardless of how many objects of object class A may exist. Furthermore, the global lock method (Global Lock1) 216 and the Unlock method 221 are methods of the "Object" object class, which is the object class at the top of the object class hierarchy in the preferred embodiment.

[0027] The Global Lock1 method 216 is used to handle requests by threads to synchronize with an object that has not yet been allocated a lock data subarray. The unlock method 221 is used to handle requests by threads to desynchronize with an object. The unlock method 221 is conventional in operation, removing the current owner of a specified object's monitor, setting the lock status to unlocked if there are no threads waiting on the specified object's monitor, and otherwise making the topmost waiting thread the owner of the specified object's monitor while leaving the lock status as locked.

[0028] Fig. 2 also shows that the "Object" object class also includes a second lock method 222, called the Lock Data CleanUp method for reclaiming the lock data subarray from an object. It should be pointed out that the three lock methods 216, 221, and 222 could be implemented as the methods of any object class known to be available in all systems using the methodology of the present invention, and do not need to be part of the "Object" object class.

[0029] Fig. 3 shows the data structure 240 for a locked object in a preferred embodiment of the present invention. This is also the data structure for any object that has recently been locked, and thus has been allocated a lock data subarray. A locked object of object class A has an object handle 242 that includes a pointer 244 to the methods for the object and a pointer 246 to a data array 208 for the object.

[0030] The method pointer 244 of an object that has recently been locked points to a object-specific version of the Virtual Function Table 250 (VFT, A-01). Each object that has an allocated lock data subarray has a specific VFT (VFT, A-01) 250 that includes: (A) pointers 212 to each of the methods 214 of the object class; (B) a pointer 256 to a local object-specific locking procedure (Local Lock2) 260 used for synchronizing an object to a thread; (C) a pointer 217 to a special Class Object 218; and (D) a pointer 220 to an unlock method for releasing the lock monitor. There is one Class Object 218 for each defined object class, and the Class Object includes a permanently allocated lock data subarray (otherwise referred to as a lock monitor) 219.

[0031] Local object-specific locking procedure 260 is used to service lock requests on objects that are locked and objects that have recently been locked. The local object-specific locking procedure 260 includes as private data a lock data subarray 249 for storing lock data. The local object-specific locking procedure has instructions for updating that object's stored lock data. More specifically, the local object-specific locking procedure 260 has an object-specific locking method (Local Lock2) for synchronizing the corresponding object to a thread and for storing the corresponding lock information in the object's lock data subarray 249.

[0032] The lock data subarray 249 includes an object lock status indicator, a lock owner value, and a list header and list tail for a list of waiters (i.e., threads waiting to synchronize with the object). In the preferred embodiment, the lock status indicator includes a first flag value (the Lock flag) that indicates whether the object is locked or unlocked, and a second flag value (the NotRecentlyLocked flag) that indicates whether or not the object has been recently locked. The NotRecentlyLocked flag is set (to True) if the object has not been recently locked. In the preferred embodiment, the NotRecentlyLocked flag is set by the Lock Data CleanUp method, and is cleared by the Global Lock1 and Local Lock2 methods.

The Object Locking Methodology

[0033] Each computer system, such as a client computer 102, has many objects, each having an associated object class. Every object is said to be an instance of its associated object class. Each object class inherits properties from its superclass, and every object class is a subclass of a top level object class called the "Object" object class.

[0034] For each object class that exists in a particular address space, there is a virtual function table (VFT) that contains a list of all the methods (i.e., executable procedures) associated with the object class as well as a pointer to each of those methods. As shown in Fig. 2, the VFT for each object class also includes a reference to the Global Lock1 method, which in the preferred embodiment is a method associated with the "Object" object class. Whenever an object has not been allocated a lock data subarray, its method pointer points to the default VFT for the object's object class.

[0035] In accordance with a first preferred embodiment of the present invention, each object class has an associated virtual function table mentioned above as the first VFT, and sometimes herein referred to as "the primary VFT." Further, for each object that has been recently locked, the system creates an object-specific virtual function table (VFT). The object-specific VFT references a local lock method, Local Lock2 260, that is different from the global lock method, Global Lock1 216, referenced by the primary VFT.

[0036] Tables 1, 2, 3 and 4 contain pseudocode representations of the Global Lock1, Local Lock2, and Lock Data CleanUp software routines relevant to embodiments of the present invention. The pseudocode used in these appen-

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.