(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
4 May 2006 (04.05.2006)

PCT

(10) International Publication Number
WO 2006/045704 A1

(51) International Patent Classification:
G06F 9/46 (2006.01)

(21) International Application Number:
PCT/EP2005/055240

(22) International Filing Date: 13 October 2005 (13.10.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
10/974,514 27 October 2004 (27.10.2004) US

(71) Applicant *(for all designated States except US)*: **INTERNATIONAL BUSINESS MACHINES CORPORATION** [US/US]; New Orchard Road, Armonk, New York 10504 (US).

(71) Applicant *(for MG only)*: **IBM UNITED KINGDOM LIMITED** [GB/GB]; PO Box 41 North Harbour, Portsmouth Hampshire PO6 3AU (GB).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **MCKENNEY, Paul** [US/US]; 1975 NW Albion Court, Beverton, Oregon 97006 (US). **RUSSELL, Paul** [AU/AU]; 1/7 Adams Street, Queanbeyan, New South Wales (AU). **SARMA, Dipankar** [IN/IN]; 303E R.J. Garden Aparments 17th E., Cross, Indiranager 2nd Stage, Bangalore, Karnatake 560038 (IN).

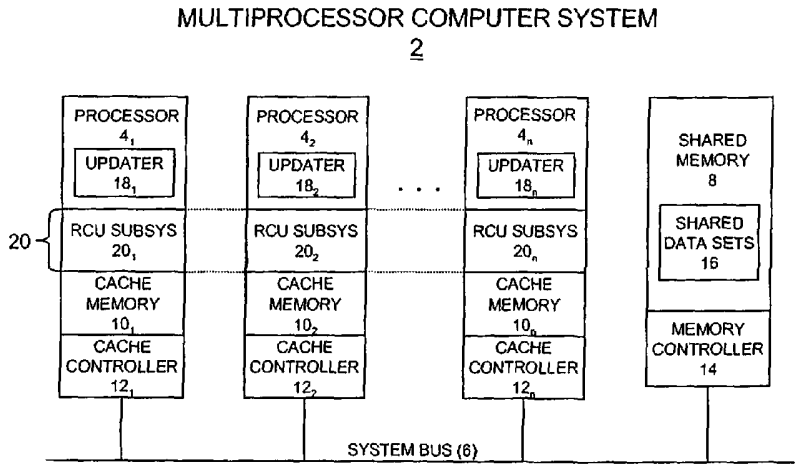
(74) Agent: **WALDNER, Philip**; IBM United Kingdom Limited, Intellectual Property Law, Hursley Park, Winchester Hampshire SO21 2JN (GB).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT,

[Continued on next page]

(54) Title: READ-COPY UPDATE GRACE PERIOD DETECTION WITHOUT ATOMIC INSTRUCTIONS THAT GRACEFULLY HANDLES LARGE NUMBERS OF PROCESSORS



(57) Abstract: A method, system and computer program product for detecting a grace period without atomic instructions in a read-copy update subsystem or other processing environment that requires deferring removal of a shared data element until pre-existing references to the data element are removed. Detection of the grace period includes establishing a token to be circulated between processing entities sharing access to the data element. A grace period elapses whenever the token makes a round trip through the processing entities. A distributed indicator associated with each processing entity indicates whether there is a need to perform removal processing on any shared data element. The distributed indicator is processed at each processing entity before the latter engages in token processing. Token processing is performed only when warranted by the distributed indicator. In this way, unnecessary token processing can be avoided when the distributed indicator does not warrant such processing.

WO 2006/045704 A1

RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *with international search report*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

READ-COPY UPDATE GRACE PERIOD DETECTION WITHOUT ATOMIC INSTRUCTIONS THAT GRACEFULLY HANDLES LARGE NUMBERS OF PROCESSORS

BACKGROUND OF THE INVENTION

Field of the Invention

The present invention relates to computer systems and methods in which data resources are shared among concurrent data consumers while preserving data integrity and consistency relative to each consumer. More particularly, the invention concerns improvements to a mutual exclusion mechanism known as "read-copy update," in which lock-free data read operations run concurrently with data update operations.

Description of the Prior Art

By way of background, read-copy update is a mutual exclusion technique that permits shared data to be accessed for reading without the use of locks, writes to shared memory, memory barriers, atomic instructions, or other computationally expensive synchronization mechanisms, while still permitting the data to be updated (modify, delete, insert, etc.) concurrently. The technique is well suited to multiprocessor computing environments in which the number of read operations (readers) accessing a shared data set is large in comparison to the number of update operations (updaters), and wherein the overhead cost of employing other mutual exclusion techniques (such as locks) for each read operation would be high. By way of example, a network routing table that is updated at most once every few minutes but searched many thousands of times per second is a case where read-side lock acquisition would be quite burdensome.

The read-copy update technique implements data updates in two phases. In the first (initial update) phase, the actual data update is carried out in a manner that temporarily preserves two views of the data being updated. One view is the old (pre-update) data state that is maintained for the benefit of operations that may be currently referencing the data. The other view is the new (post-update) data state that is available for the benefit of operations that access the data following the update. In the second (deferred update) phase, the old data state is removed following a "grace period" that is long enough to ensure that all executing operations will no longer maintain references to the pre-update data.

Figs. 1A-1D illustrate the use of read-copy update to modify a data element B in a group of data elements A, B and C. The data elements A, B, and C are arranged in a singly-linked list that is traversed in acyclic fashion, with each element containing a pointer to a next element in the list (or a NULL pointer for the last element) in addition to storing some item of data. A global pointer (not shown) is assumed to point to data element A, the first member of the list. Persons skilled in the art will appreciate that the data elements A, B and C can be implemented using any of a variety of conventional programming constructs, including but not limited to, data structures defined by C-language "struct" variables.

It is assumed that the data element list of Figs. 1A-1D is traversed (without locking) by multiple concurrent readers and occasionally updated by updaters that delete, insert or modify data elements in the list. In Fig. 1A, the data element B is being referenced by a reader r1, as shown by the vertical arrow below the data element. In Fig. 1B, an updater u1 wishes to update the linked list by modifying data element B. Instead of simply updating this data element without regard to the fact that r1 is referencing it (which might crash r1), u1 preserves B while generating an updated version thereof (shown in Fig. 1C as data element B') and inserting it into the linked list. This is done by u1 acquiring a spinlock, allocating new memory for B', copying the contents of B to B', modifying B' as needed, updating the pointer from A to B so that it points to B', and releasing the spinlock. All subsequent (post update) readers that traverse the linked list, such as the reader r2, will thus see the effect of the update operation by encountering B'. On the other hand, the old reader r1 will be unaffected because the original version of B and its pointer to C are retained. Although r1 will now be reading stale data, there are many cases where this can be tolerated, such as when data elements track the state of components external to the computer system (e.g., network connectivity) and must tolerate old data because of communication delays.

At some subsequent time following the update, r1 will have continued its traversal of the linked list and moved its reference off of B. In addition, there will be a time at which no other reader process is entitled to access B. It is at this point, representing expiration of the grace period referred to above, that u1 can free B, as shown in Fig. 1D.

Figs. 2A-2C illustrate the use of read-copy update to delete a data element B in a singly-linked list of data elements A, B and C. As shown in Fig. 2A, a reader r1 is assumed be currently referencing B and an

updater u1 wishes to delete B. As shown in Fig. 2B, the updater u1 updates the pointer from A to B so that A now points to C. In this way, r1 is not disturbed but a subsequent reader r2 sees the effect of the deletion. As shown in Fig. 2C, r1 will subsequently move its reference off of B, allowing B to be freed following expiration of the grace period.

In the context of the read-copy update mechanism, a grace period represents the point at which all running processes having access to a data element guarded by read-copy update have passed through a "quiescent state" in which they can no longer maintain references to the data element, assert locks thereon, or make any assumptions about data element state. By convention, for operating system kernel code paths, a context (process) switch, an idle loop, and user mode execution all represent quiescent states for any given CPU (as can other operations that will not be listed here).

In Fig. 3, four processes 0, 1, 2, and 3 running on four separate CPUs are shown to pass periodically through quiescent states (represented by the double vertical bars). The grace period (shown by the dotted vertical lines) encompasses the time frame in which all four processes have passed through one quiescent state. If the four processes 0, 1, 2, and 3 were reader processes traversing the linked lists of Figs. 1A-1D or Figs. 2A-2C, none of these processes having reference to the old data element B prior to the grace period could maintain a reference thereto following the grace period. All post grace period searches conducted by these processes would bypass B by following the links inserted by the updater.

There are various methods that may be used to implement a deferred data update following a grace period, including but not limited to the use of callback processing as described in commonly assigned U.S. Patent No. 5,727,209, entitled "Apparatus And Method For Achieving Reduced Overhead Mutual-Exclusion And Maintaining Coherency In A Multiprocessor System Utilizing Execution History And Thread Monitoring." The contents of U.S. Patent No. 5,727,209 are hereby incorporated herein by this reference.

The callback processing technique contemplates that an updater of a shared data element will perform the initial (first phase) data update operation that creates the new view of the data being updated, and then specify a callback function for performing the deferred (second phase) data update operation that removes the old view of the data being updated. The updater will register the callback function (hereinafter referred to

as a "callback") with a read-copy update subsystem so that it can be executed at the end of the grace period. The read-copy update subsystem keeps track of pending callbacks for each processor and monitors per-processor quiescent state activity in order to detect when each processor's current grace period has expired. As each grace period expires, all scheduled callbacks that are ripe for processing are executed.

The successful implementation of read-copy update requires efficient mechanisms for deducing the length of a grace period. One important class of implementations passes a grace period token from one processor to the next to signify that the end of a grace period has been reached for the processor owning the token. The grace period token can be a distinguished value that is expressly passed between processors. However, two memory write accesses are required when using this technique -- one to remove the token from its current owner and another to pass the token to its new owner. A more efficient way of handling the grace period token is to pass it implicitly using per-processor quiescent state counters and associated polling mechanisms. According to this technique, whenever a processor passes through a quiescent state, its polling mechanism inspects the quiescent state counter of a neighboring processor to see if the neighbor's counter has changed since the current processor's last grace period. If it has, the current processor determines that a new grace period has elapsed since it last had the token. It executes its pending callbacks and then changes its quiescent state counter to an incrementally higher value than that of its neighbor. The next processor then sees this processor's changed counter value, processes its pending callbacks, and increments its own counter. This sequence continues, with the grace period token ultimately making its way through all of the processors in round-robin fashion.

Regardless of how the grace period token is implemented, each processor only processes callbacks when it receives the token. Insofar as the grace period token must travel through all other processors before reaching the processor that is the current holder, the current processor is always guaranteed that the other processors have passed through a quiescent state since the last time the current processor owned the token, thus ensuring that a grace period has elapsed.

Because grace period detection using token manipulation consumes processor cycles as the processors pass through their quiescent states, it is undesirable to incur such overhead unless there are pending callbacks

in the read-copy update subsystem. For that reason, efficient token-based read-copy update implementations use a shared indicator (i.e., a global variable) that is tested before grace period token processing to determine if the read-copy update subsystem is idle. If it is, the grace period token does not need to be passed and the associated processing overhead can be avoided. The shared indicator is typically a count of the number of pending callbacks. Whenever a callback is registered at a given processor, the shared indicator is manipulated to reflect the new callback. Thereafter, when that callback is processed, the shared indicator is again manipulated to reflect the removal of the callback from the read-copy update subsystem.

A disadvantage of using a shared indicator to test for the existence of pending callbacks is that atomic instructions, locks or other relatively expensive mutual exclusion mechanisms must be invoked each time the shared indicator is manipulated in order to synchronize operations on the indicator by multiple processors. Moreover, conventional hardware caching of the shared indicator by each processor tends to result in communication cache misses and cache line bouncing. In the case of a bitmap indicator, a further disadvantage is that a large number of processors cannot be gracefully accommodated.

It is to solving the foregoing problems that the present invention is directed. In particular, what is required is a new read-copy update grace period detection technique that avoids unnecessary grace period token processing without incurring the overhead of a shared indicator of pending callback status.

Summary of the Invention

The foregoing problems are solved and an advance in the art is obtained by a method, system and computer program product for detecting a grace period without atomic instructions in a read-copy update subsystem or other processing environment that requires deferring removal of a shared data element until pre-existing references to the data element are removed. Grace period detection includes establishing a token to be circulated between processing entities sharing access to the shared data element. A grace period can be determined to elapse whenever the token makes a round trip through the processing entities. A distributed indicator is associated with each of the processing entities that is indicative of whether there is a need to perform removal processing on the data element or on other data elements shared by the processing entities

(e.g., whether there are pending callbacks warranting callback processing if the invention is implemented in a callback-based read-copy update system). The distributed indicator is processed at each of the processing entities before engaging in token processing at the processing entities. Token processing is performed at the processing entities only when warranted by the distributed indicator. In this way, unnecessary token processing can be avoided when the distributed indicator does not warrant such processing.

In exemplary embodiments of the invention, the distributed indicators are stored as local variables in the cache memories associated with the processing entities (and replicated from one cache memory to another during the course of processing via conventional cache coherence mechanisms). In such embodiments, the distributed indicators can represent different kinds of information depending on design preferences. For example, the distributed indicators can alternatively represent the number of processing entities that have pending requests to perform updates to data elements shared by the processing entities, the total number of updates, or a bitmap identifying the processing entities having pending update requests.

The propagation of changes made to the distributed indicators by the various processing entities can also be performed in different ways according to design preferences. In exemplary embodiments, the processing entities periodically consult a distributed indicator maintained by a neighboring processing entity, and adjust the indicator as necessary to reflect changes in data element removal request activity (e.g., callback registrations) at the current processing entity. Whether there has been a change in data element removal request activity can include determination of various factors, such as whether there are a threshold number of pending data element removal requests at one of the processing entities to warrant circulation of the token. Alternatively, such determination could be based on whether there are any pending data element removal requests at one of the processing entities.

Brief Description of the Drawings

The foregoing and other features and advantages of the invention will be apparent from the following more particular description of exemplary embodiments of the invention, as illustrated in the accompanying Drawings, in which:

Figs. 1A-1D are diagrammatic representations of a linked list of data elements undergoing a data element replacement according to a conventional read-copy update mechanism;

Figs. 2A-2C are diagrammatic representations of a linked list of data elements undergoing a data element deletion according to a conventional read-copy update mechanism;

Fig. 3 is a flow diagram illustrating a grace period in which four processes pass through a quiescent state;

Fig. 4 is a functional block diagram showing a multiprocessor computing system that represents one exemplary environment in which the present invention can be implemented;

Fig. 5 is a functional block diagram showing a read-copy update subsystem implemented by each processor in the multiprocessor computer system of Fig. 4;

Fig. 6 is a functional block diagram showing a cache memory associated with each processor in the multiprocessor computer system of Fig. 4;

Fig. 7 is a table showing exemplary quiescent state counter values in a hypothetical four-processor data processing system implementing read-copy update;

Fig. 8 is a functional block diagram showing the four processors of Fig. 7 as they pass a grace period token from time to time during read-copy update processing;

Fig. 9 is a flow diagram showing the manipulation of a distributed callback indicator implemented as a count of processors having pending callbacks;

Fig. 10 is a table showing exemplary quiescent state counter values and distributed callback indicator values in a hypothetical four-processor data processing system implementing read-copy update;

Fig. 11 is a functional block diagram showing the four processors of Fig. 10 as they pass a grace period token from time to time during read-copy update processing;

Fig. 12 is a flow diagram representing a modification of the flow diagram of Fig. 9;

Fig. 13 is a flow diagram showing the manipulation of a distributed callback indicator implemented as a count of pending callbacks;

Fig. 14 is a flow diagram showing the manipulation of a distributed callback indicator implemented as a bitmap identifying processors having pending callbacks; and

Fig. 15 is a diagrammatic illustration of storage media that can be used to store a computer program product for implementing read-copy update grace period detection functions in accordance with the invention.

Detailed Description of Exemplary Embodiments

Turning now to the figures, wherein like reference numerals represent like elements in all of the several views, Fig. 4 illustrates an exemplary computing environment in which the present invention may be implemented. In particular, a symmetrical multiprocessor (SMP) computing system 2 is shown in which multiple processors 4_1, 4_2 . . . 4_n are connected by way of a common bus 6 to a shared memory 8. Respectively associated with each processor 4_1, 4_2 . . . 4_n is a conventional cache memory 10_1, 10_2 . . . 10_n and a cache controller 12_1, 12_2 . . . 12_n. A conventional memory controller 14 is associated with the shared memory 8. The computing system 2 is assumed to be under the management of a single multitasking operating system adapted for use in an SMP environment.

It is further assumed that update operations executed within kernel or user mode processes, threads, or other execution contexts will periodically perform updates on shared data sets 16 stored in the shared memory 8. Reference numerals 18_1, 18_2 . . . 18_n illustrate individual data update operations (updaters) that may periodically execute on the several processors 4_1, 4_2 . . . 4_n. As described by way of background above, the updates performed by the data updaters 18_1, 18_2 . . . 18_n can include modifying elements of a linked list, inserting new elements into the list, deleting elements from the list, and many other types of operations. To facilitate such updates, the several processors 4_1, 4_2 . . . 4_n are programmed to implement a read-copy update (RCU) subsystem 20, as by periodically executing respective read-copy update instances 20_1, 20_2 . . . 20_n as part of their operating system functions. Although not illustrated in the drawings, it will be appreciated that the processors 4_1, 4_2 . . . 4_n

also execute read operations on the shared data sets 16. Such read operations will typically be performed far more often than updates, insofar as this is one of the premises underlying the use of read-copy update.

As shown in Fig. 5, each of the read-copy update subsystem instances 20_1, 20_2 . . . 20_n includes a callback registration component 22. The callback registration component 22 serves as an API (Application Program Interface) to the read-copy update subsystem 20 that can be called by the updaters 18_2 . . . 18_n to register requests for deferred (second phase) data element updates following initial (first phase) updates performed by the updaters themselves. As is known in the art, these deferred update requests involve the removal of stale data elements, and will be handled as callbacks within the read-copy update subsystem 20. Each of the read-copy update subsystem instances 20_1, 20_2 . . . 20_n additionally includes a quiescent state counter manipulation and polling mechanism 24 (or other functionality for passing a token), together with a callback processing system 26. Note that the functions 24 and 26 can be implemented as part of a kernel scheduler, as is conventional.

The cache memories 10_1, 10_2 . . . 10_n associated with the processors 4_1, 4_2 . . . 4_n respectively store quiescent state counters 28_1, 28_2 . . . 28_n and one or more callback queues 30_1, 30_2 . . . 30_n. The quiescent state counters 28_1, 28_2 . . . 28_n are managed by the counter manipulation and polling mechanism 24 (a token manipulator) for the purpose of passing a grace period token among the processors 4_1, 4_2 . . . 4_n. It will be appreciated that if some other form of token passing is used, the quiescent state counters 28_1, 28_2 . . . 28_n will not be required. The callback queues 30_1, 30_2 . . . 30_n are appended (or prepended) with new callbacks as such callbacks are registered with the callback registration component 22. The callback processing system 26 is responsible for executing the callbacks referenced on the callback queues 30_1, 30_2 . . . 30_n, and then removing the callbacks as they are processed.

Figs. 7 and 8 illustrate how the quiescent state counters 28_1, 28_2 . . . 28_n can be used to pass a grace period token between processors in an exemplary four processor system as the processors pass through quiescent states. Each column in Fig. 7 shows exemplary values for all processor quiescent state counters at a given point in time. The shaded cells indicate that the corresponding processor is the owner of the grace period token. In each case, the owner is the processor whose counter has the

smallest value and whose neighbor has a counter value representing a discontinuity relative to the token owner's counter value.

The token passing technique represented by Figs. 7 and 8 is known in the art and these figures are therefore labeled as "Prior Art." As described by way of background above, a given processor checks to see if it owns the grace period token by referring to the quiescent state counter maintained by one of neighbors (e.g., that processor whose processor number is one greater than the current processor, modulo (%) the number of processors). If the neighbor's quiescent state counter has not changed since the current processor's last grace period (i.e., there is no discontinuity in the counter values), the current processor determines that a new grace period has not yet elapsed and resumes normal processing. If the neighbor's counter has changed since the current processor's last grace period (i.e., there is a discontinuity in the counter values), the current processor determines that a new grace period has elapsed. It processes its pending callbacks and increments its own quiescent state counter to one greater than the neighbor's value, thereby moving the discontinuity in counter values to itself. By way of example, at time t=0 in Figs. 7, processor 3 that has the lowest quiescent state counter value (1) sees a discontinuous counter value (4) at processor 0. This signifies to processor 3 that there have been three (4-1) quiescent states experienced by its peer processors since processor 3's last grace period. Processor 3 thus concludes that a new grace period has elapsed and that it now has the grace period token. It performs callback processing and sets its quiescent state counter value to 4+1=5. At time t=1, processor 2, having a quiescent state counter value of 2, now sees the discontinuous counter value 5 at processor 0. It determines that it has the grace period token, performs callback processing, and sets its counter value to 5+1=6. Continuing this sequence, processor 1 obtains the grace period token at time t=2 and processor 0 obtains the token at time t=3. At time t=4, the token returns to processor 3 and the pattern repeats. As additionally shown in Fig. 8, processor 3 will obtain the token (T) at times t= 0, 4 and 8, processor 2 will obtain the token at times t=1 and 5, processor 3 will obtain the token at times t=2 and 6, and processor 0 will obtain the token at times t=3 and 7.

As also described by way of background above, prior art implementations of read-copy update seek to avoid unnecessary token processing by manipulating a global variable that serves as a shared indicator of whether there are pending callbacks in the read-copy update subsystem that require processing. For example, as disclosed in P.

McKenney et al., "Read Copy Update," Ottawa Linux Symposium (2002), a Linux implementation of read-copy update known as "rcu-sched" uses a shared variable "**rcu_pending**" that represents a count of the number of pending callbacks in the read-copy update subsystem. The Linux atomic increment primitive "**atomic_inc**" is invoked to increment **rcu_pending** when a new callback is registered by way of the function call "**atomic_inc(&rcu_pending).**" The Linux atomic decrement primitive "**atomic_dec**" is then invoked to decrement **rcu_pending** after the callback is processed by way of the function call "**atomic_dec(&rcu_pending).**" It should also be pointed out that "rcu-sched" is an example of a read-copy update implementation that uses a counter-based, grace period token passing scheme as shown in Figs. 7 and 8.

In order to avoid the disadvantages associated with the use of atomic operations (or other concurrency control mechanisms) to increment and decrement a shared indicator of callback pendency, the present invention proposes an alternative approach. As shown in Fig. 6, a distributed callback indicator 32 can be maintained in the cache memory 10 of each of the processors 4_1, 4_2 . . . 4_n and manipulated as a local variable to reflect changes in the read-copy update subsystem 20. Each distributed callback indicator 32 provides a representation of the state of the read-copy update subsystem 20. An associated callback indicator handling mechanism 34 (also shown in Fig. 6) within each of the read-copy update subsystem instances 20_1, 20_2 . . . 20_n can then consult the local distributed callback indicator 32 to determine whether grace period token processing is required. The local distributed callback indicator 32 may show that the read-copy update subsystem is idle, in which case the token does not need to be passed. On the other hand, the local distributed callback indicator 32 may show that there are callbacks pending in the read-copy update subsystem, and that grace period token processing is required at the current processor.

In order to keep the distributed callback indicators 32 current as conditions change within the read-copy update subsystem 20, a propagation technique that is somewhat analogous to the grace period token passing scheme of Figs. 7 and 8 may be used. Other implementations would also be possible. According to the propagation technique, as each of the processors 4_1, 4_2 . . . 4_n passes through a quiescent state, its callback indicator handling mechanism 34 consults the distributed callback indicator 32 of a neighbor processor and adjusts its own local callback indicator according to the neighbor's value, coupled with consideration of

the local callback history since the current processor's last grace period.

In one embodiment of the invention, the distributed callback indicator 32 is implemented as a per-processor counter of the number of processors having pending callbacks. These processors may be referred to as "callback processors," and the distributed callback indicator 32 may be thought of as a callback processor counter. To manipulate this counter, a processor checks to see if there has been any change in its local callback state since this processor's last grace period. If no change has occurred, the current processor's counter will be set to the same value as a neighbor processor's counter. If a processor's callback history shows that no local callbacks were registered the last time the grace period token left this processor, but a requisite number of new local callbacks have been registered since the last grace period, the current processor's counter will be incremented to one higher than the value of the neighbor processor's counter. If a processor's callback history shows that local callbacks were registered the last time the grace period token left this processor, but a requisite number of new local callbacks have not been registered since the last grace period, the current processor's counter will be decremented so as to be one lower than the value of the neighbor processor's counter.

In a second embodiment of the invention, the distributed callback indicator 32 is implemented to track an indication of the total number of pending callbacks. In that case, the distributed callback indicator 32 can be thought of as a callback counter. To manipulate this counter, a processor compares the number of local callbacks that have been registered since this processor's last grace period to the number of local callbacks that were registered the last time the grace period token left the processor. The current processor's counter is set to the value of a neighbor processor's counter with an adjustment to reflect the net gain or loss of local callbacks.

In a third embodiment of the invention, the distributed callback indicator 32 is implemented as a bitmap identifying processors that have pending callbacks. To manipulate the bitmap, a processor determines if there are a requisite number of local callbacks that have been registered since the last time the grace period token left this processor. If there are, the current processor's bitmap is set to correspond to a neighbor processor's bitmap, but with the current processor's bit set to 1. Otherwise, if a requisite number of local callbacks have not been

registered since the last grace period, the current processor' bit value in the bit map is set to zero. One disadvantage of this implementation is that it does not gracefully handle large numbers of processors due to need to process correspondingly large bitmaps.

Fig. 9 illustrates an exemplary sequence of processing steps that may be performed according to the first above-described embodiment in which the distributed callback indicator 32 is a count of the number of processors 4_1, 4_2 . . . 4_n that have pending callbacks. The process of Fig. 9 uses a per-processor local variable called "**cbcpus**" (shorthand for "callback cpus") as the distributed callback indicator. This variable is a count of processors having callbacks needing processing. Another per-processor local variable, called "**lastcbs**" (shorthand for "last callbacks"), is a flag indicating whether the current processor had callbacks registered the last time the grace period token left this processor. A third per-processor variable, called "**numcbs**" (shorthand for "number of callbacks") is a count of the number of callbacks registered at the current processor since the last grace period.

In step 40 of Fig. 9, the nth processor's callback indicator handling mechanism 34 obtains the value of **cbcpus** of the processor n+1 (processor n-1 could also be used depending on the desired propagation direction). In step 42, processor n determines if there are any new callbacks (**numcbs**) that meet the criteria for starting a grace period. In some cases, the presence of a single callback will satisfy this criteria. In other cases, it may be desirable to batch process callbacks by establishing a callback threshold specifying the number of callbacks necessary to start a grace period, and an elapsed time threshold that triggers callback processing even if the callback threshold is not reached. If in step 42 there are new callbacks requiring processing, then in step 44 the current processor's value of **cbcpus** is set to one greater than the neighbor processor's value of **cbcpus**, less the current processor's value of **lastcbs**. The value of **lastcbs** is then set to 1 in step 46 if and only if the callbacks on the current processor meet the criteria for starting a grace period. If in step 42 there are no new callbacks requiring processing, then in step 48 the current processor's value of **cbcpus** is set equal to the neighbor processor's value of **cbcpus**, less the current processor's value of **lastcbs**. The value of **lastcbs** is then set to 0 in step 50 if and only if there are no new callbacks on the current processor that meet the criteria for starting a grace period.

As each processor performs the foregoing processing while passing through a quiescent state, changes due to the registration of new callbacks or the processing of old callbacks will be quickly reflected by each of the distributed callback indicators (**cbcpus** in this example). By testing the propagated distributed callback indicator at each processor, potentially expensive token processing can be avoided when there are not enough callbacks warranting grace period token circulation. The table of Fig. 10 is illustrative of such processing in an exemplary four-processor system. Fig. 10 is based on Fig. 7 but shows, for each processor 0, 1, 2, and 3, both a grace period token on the left side of each table element and a distributed callback indicator (**cbcpus** in this example) on the right side of each table element. The shaded cells again indicate that the corresponding processor is the owner of the grace period token. In each case, the owner is the processor whose quiescent state counter has the smallest value and whose neighbor has a counter value representing a discontinuity relative to the token owner's counter value.

In Fig. 10, processor 3 receives the grace period token from processor 0. However, no token processing takes place because processor 3's distributed callback indicator has a value of 0. In the current example in which the distributed callback indicator 32 is a count of callback processors (**cbcpus**), the 0 value means there are no processors having a requisite number of callbacks warranting processing. Processor 3 thus determines that the read-copy update subsystem for this group of processors is idle. At time t=1 in Fig. 10, processor 2 determines that it has had new callback activity and sets its distributed callback indicator to a value of 1. Processor 3 is unaffected (since it only looks to processor 0 for callback indicator activity according to the current example) and again performs no grace period token processing. At time t=2, processor 2's distributed callback indicator value is propagated to processor 1. Processor 3 is unaffected and again performs no grace period token processing. At time t=3, processor 1's distributed callback indicator value has propagated to processor 0. Processor 3 is unaffected and again performs no grace period token processing. At time t=4, processor 0's distributed callback indicator value has been propagated to processor 3, causing it to perform grace period token processing and pass the token to processor 2. At time t=5, processor 2 has performed grace period token processing and passed the token to processor 1. At time t=6, processor 1 has performed grace period token processing and passed the token to processor 0. In addition, it is assumed that processor 2 has determined that its callbacks have been processed and set its distributed callback indicator to 0. At time t=7, processor 0 has performed grace

period token processing and passed the token to processor 3. In addition, processor 2's distributed callback indicator has been propagated to processor 1. At time t=8, processor 3 has performed grace period token processing and passed the token to processor 2. In addition, processor 1's distributed callback indicator has been propagated to processor 0. Assuming no new callbacks are registered in the system of Fig. 9, the grace period token will now idle at processor 2 because its distributed callback indicator is 0.

Fig. 11 summarizes the foregoing processing. It shows that processor 3 will obtain the token (T) at times t= 0, 7. The token will then idle at processor 3 during times t=1, 2 and 3. Processor 2 will then obtain the token at times 4, 8. Processor 1 will obtain the token at time t=5. Processor 0 will obtain the processor at time t=6.

Turning now to Fig. 12, an alternative to the distributed callback indicator processing of Fig. 9 is shown. According to this alternative approach, step 42a (corresponding to step 42 of Fig. 9) inquires whether **numbcbs** is nonzero, without regard to whether a threshold has been reached. Step 46a (corresponding to step 46 of Fig. 9) sets **lastcbs** to 1 if and only **numcbs** is greater than 0. Step 50a (corresponding to step 50 of Fig. 9) sets **lastcbs** to 0 if and only **numcbs** is 0. The advantage of this alternative approach is that it permits processors with only a few callbacks to "piggyback" their callback processing needs onto another processor's grace period token circulation and keep the token moving. The disadvantage is that additional grace period detection operations can result.

Fig. 13 illustrates an exemplary sequence of processing steps that may be performed according to the second above-described embodiment in which the distributed callback indicator 32 is a count of the number of pending callbacks. The process of Fig. 13 uses a per-processor local variable called "**cbspen**" (shorthand for "callbacks pending") as the distributed callback indicator. Another per-processor local variable, called "**lastcbs**" (shorthand for "last callbacks"), is a value indicating the number of callbacks that the current processor had registered the last time the grace period token left this processor. A third per-processor variable, called "**numcbs**" (shorthand for "number of callbacks") is a count of the number of callbacks registered at the current processor since the last grace period.

In step 60 of Fig. 13, the nth processor's callback indicator handling mechanism 34 obtains the value of **cbspen** of the processor n+1 (processor n-1 could also be used depending on the desired propagation direction). In step 62, the current processor's value of **cbspen** is set to the neighbor processor's value of **cbspen**, plus the current processor's value of **numcbs**, less the current processor's value of **lastcbs**. The value of **lastcbs** is then set to **numcbs** in step 66.

Fig. 14 illustrates an exemplary sequence of processing steps that may be performed according to the third above-described embodiment in which the distributed callback indicator 32 is a bit map showing which processors have pending callbacks. The process of Fig. 14 uses a per-processor local bitmap variable called "**cbcpumap**" (shorthand for "callback cpu map") as the distributed callback indicator. Another per-processor local variable, called "**numcbs**" (shorthand for "number of callbacks") is a count of the number of callbacks registered at the current processor since the last grace period.

In step 80 of Fig. 14, the nth processor's callback indicator handling mechanism 34 obtains the value of **cbcpumap** of the processor n+1 (processor n-1 could also be used depending on the desired propagation direction). In step 82, processor n determines if there are any new callbacks (**numcbs**) registered at this processor that satisfy some established threshold (e.g., as discussed above relative to Fig. 9). If in step 82 there are new callbacks requiring processing, then in step 84 the current processor's **cpcumap** is set equal to that of processor n+1, but the nth bit of **cbcpumap** is set to 1. If in step 82 there are no new callbacks requiring processing, then in step 86 the current processor's value of **cpcpumap** is set equal to that of processor n+1, but the nth bit of **cbcpumap** is set to 0.

Accordingly, a technique for read-copy update grace period detection has been disclosed that does not require atomic instructions and which can be implemented to gracefully handle large numbers of processors. It will be appreciated that the foregoing concepts may be variously embodied in any of a data processing system, a machine implemented method, and a computer program product in which programming means are recorded on one or more data storage media for use in controlling a data processing system to perform the required functions. Exemplary data storage media for storing such programming means are shown by reference numeral 100 in Fig. 15. The media 100 are shown as being portable optical storage disks of the type

that are conventionally used for commercial software sales. Such media can store the programming means of the invention either alone or in conjunction with an operating system or other software product that incorporates read-copy update functionality. The programming means could also be stored on portable magnetic media (such as floppy disks, flash memory sticks, etc.) or on magnetic media combined with drive systems (e.g. disk drives) incorporated in computer platforms.

While various embodiments of the invention have been described, it should be apparent that many variations and alternative embodiments could be implemented in accordance with the invention. It is understood, therefore, that the invention is not to be in any way limited except in accordance with the spirit of the appended claims and their equivalents.

CLAIMS

1. A method for detecting a grace period for deferring removal of a shared data element until pre-existing references to the data element are removed, comprising:

establishing a token to be circulated between processing entities sharing access to said data element;

determining that said grace period has elapsed when said token makes a round trip through said processing entities;

associating a distributed indicator with each of said processing entities that is indicative of whether there is a need to perform removal processing on said data element or on other data elements shared by said processing entities;

processing said distributed indicator at each of said processing entities before engaging in token processing at said processing entities; and

performing token processing at said processing entities only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

2. A method in accordance with claim 1 wherein each of said distributed indicators is stored as a local variable in a cache memory associated with one of said processing entities.

3. A method in accordance with claim 1 wherein a distributed indicator represents a number of said processing entities that have pending requests to remove data elements shared by said processing entities or a number of pending requests to remove data elements shared by said processing entities.

4. A method in accordance with claim 1, 2 or 3 wherein said distributed indicators represent a bit map identifying said processing entities that have pending requests to remove data elements shared by said processing entities.

5. A method in accordance with any one of claims 1 to 4 wherein a change made to one of said distributed indicators at one of said processing entities is propagated to other ones of said processing entities.

6. A method in accordance with any one of claims 1 to 5 wherein a value of one of said distributed indicators at first one of said processing entities reflects a value of a second one of said distributed indicators at a neighboring second one of said processing entities adjusted as necessary to reflect data element removal request activity at said first one of said processing entities.

7. A method in accordance with Claim 1 to 6 wherein said processing of said distributed indicator includes determining whether there are a threshold number of pending data element removal requests at one of said processing entities to warrant circulation of said token or determining whether there any pending data element removal requests at one of said processing entities.

8. A method in accordance with any one of Claims 1 to 7 wherein said method is implemented as part of a read-copy update mutual exclusion technique, said data element removals are implemented by registering callbacks at said processing entities, and said distributed indicator is a distributed callback indicator reflecting callback activity in a read-copy update subsystem.

9. A data processing system having one or more processors, a memory and a communication pathway between the one or more processors and the memory, said system being adapted to detect a grace period for deferring a removal by one of said processors to a shared data element until pre-existing references to the data element maintained by other of said processors are removed, and comprising:

a token to be circulated between said processors;

a token manipulator associated with each of said processors adapted to circulate said token and determine whether said grace period has elapsed by virtue of said token making a round trip through said processors;

a distributed indicator associated with each of said processors that is indicative of whether there is a need to perform removal processing on

said data element or on other data elements shared by said processing entities;

a distributed indicator handling mechanism associated with each of said processors adapted to process said distributed indicator before token processing by said token manipulator; and

said distributed indicator handling mechanisms being further adapted to permit said token manipulators to performing token processing at said processors only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

10. A computer program product for detecting a grace period for deferring removal of a shared data element until pre-existing references to the data element are removed, comprising:

one or more data storage media;

means recorded on said data storage media for programming a data processing platform to operate as by:

establishing a token to be circulated between processors sharing access to said data element;

determining that said grace period has elapsed when said token makes a round trip through said processing entities;

associating a distributed indicator with each of said processing entities that is indicative of whether there is a need to perform removal processing on said data element or on other data elements shared by said processing entities;

processing said distributed indicator at each of said processing entities before engaging in token processing at said processing entities; and

performing token processing at said processing entities only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

30. A computer program product in accordance with Claim 21 wherein said program means are implemented as part of a read-copy update subsystem computer program product, said data element removals are implemented by registering callbacks at said processing entities, and said distributed indicator is a distributed callback indicator reflecting callback activity in said read-copy update subsystem.

11. A method for detecting a grace period in a read-copy update subsystem to determine when pending callbacks may be executed, comprising:

implementing a quiescent state counter at each of a set of processors sharing access to said a data element;

establishing a grace period token as a discontinuity in count values maintained by said quiescent state counters;

determining that said grace period has elapsed at one of said processors when said token makes a round trip through said set of processors to return to said one of said processors;

associating a distributed callback indicator as local variable in a cache memory associated with each of said processors that is indicative of whether there is a need to process said callbacks;

processing said distributed callback indicator at each of said processors before engaging in token processing at said processing entities; and

performing token processing at said processors only when warranted by said distributed callback indicator;

whereby unnecessary token processing can be avoided when said distributed callback indicator does not warrant such processing.

12. A data processing system having one or more processors, a memory and a communication pathway between the one or more processors and the memory, said system including a read-copy update subsystem adapted to detect a grace period to determine when pending callbacks may be executed, and comprising:

13. A method for detecting a grace period for deferring removal of a shared data element until pre-existing references to the data element are removed, comprising:

establishing a token to be circulated between processing entities sharing access to said data element;

determining that said grace period has elapsed when said token makes a round trip through said processing entities;

associating a distributed indicator with each of said processing entities that represents a number of said processing entities that have pending requests to remove data elements shared by said processing entities;

processing said distributed indicator at each of said processing entities before engaging in token processing at said processing entities;

said processing including modifying said distributed indicator according to a value of said distributed indicator at a neighboring one of said processing entities and a modification of said value according to (1) whether there are removal requests associated with the current grace period and there were no removal requests associated with the preceding grace period, in which case the distributed indicator is incremented, or (2) whether there are no removal requests associated with the current grace period and there were removal requests associated with the preceding grace period, in which case the distributed indicator is decremented, or (3) whether there are removal requests for both the current and preceding grace periods, or no removal requests for both the current and preceding grace periods, in which case the distributed indicator remains the same; and

performing token processing at said processing entities only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

14. A method for detecting a grace period for deferring removal of a shared data element until pre-existing references to the data element are removed, comprising:

establishing a token to be circulated between processing entities sharing access to said data element;

determining that said grace period has elapsed when said token makes a round trip through said processing entities;

associating a distributed indicator with each of said processing entities that represents a total number of pending requests to remove data elements shared by said processing entities;

processing said distributed indicator at each of said processing entities before engaging in token processing at said processing entities;

said processing including modifying said distributed indicator according to a value of said distributed indicator at a neighboring one of said processing entities and a modification of said value according to a difference between the number of removal requests added during the current grace period and the number of removal requests processed during the previous grace period; and

performing token processing at said processing entities only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

15. A method for detecting a grace period for deferring removal of a shared data element until pre-existing references to the data element are removed, comprising:

establishing a token to be circulated between processing entities sharing access to said data element;

determining that said grace period has elapsed when said token makes a round trip through said processing entities;

associating a distributed indicator with each of said processing entities that represents a bitmap of said processing entities that have pending requests to remove data elements shared by said processing entities;

processing said distributed indicator at each of said processing entities before engaging in token processing at said processing entities;

said processing including modifying said distributed indicator according to a value of said distributed indicator at a neighboring one of said processing entities and setting a bit corresponding to the evaluating processing entity according to (1) whether there are removal requests associated with the current grace period, in which case the bit is set to one, or (2) whether there are no removal requests associated with the current grace period, in which case the bit is set to zero; and

performing token processing at said processing entities only when warranted by said distributed indicator;

whereby unnecessary token processing can be avoided when said distributed indicator does not warrant such processing.

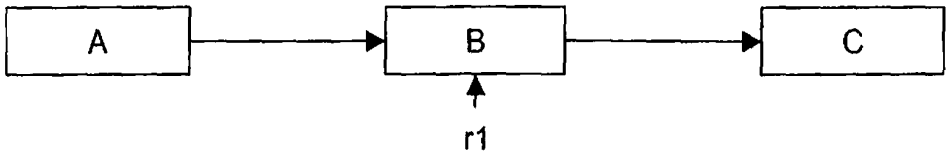


FIG. 1A (PRIOR ART)

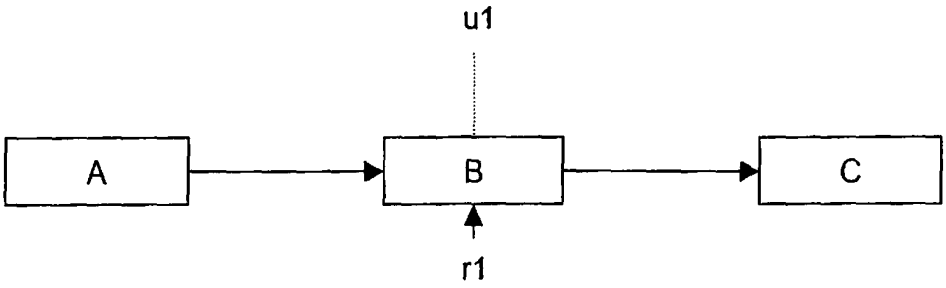


FIG. 1B (PRIOR ART)

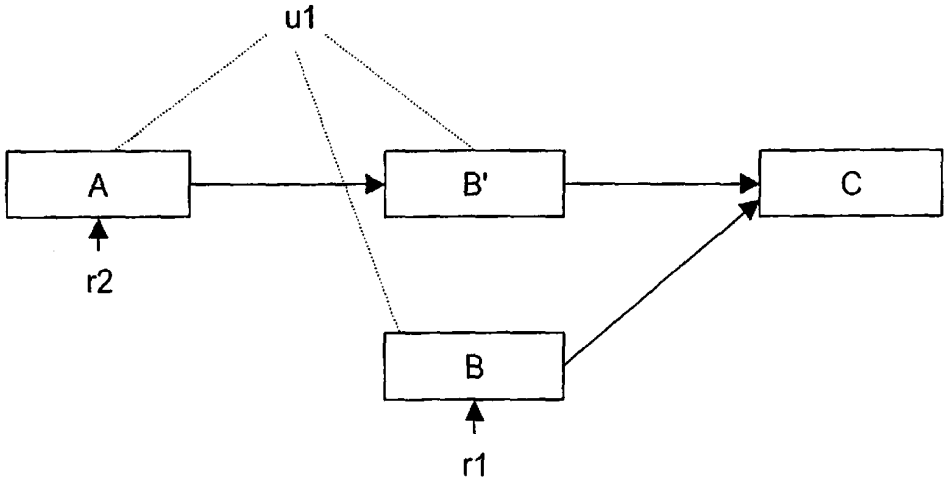


FIG. 1C (PRIOR ART)

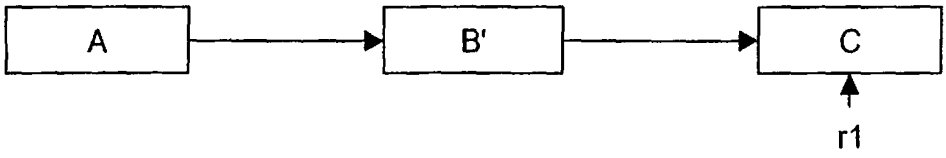


FIG. 1D (PRIOR ART)

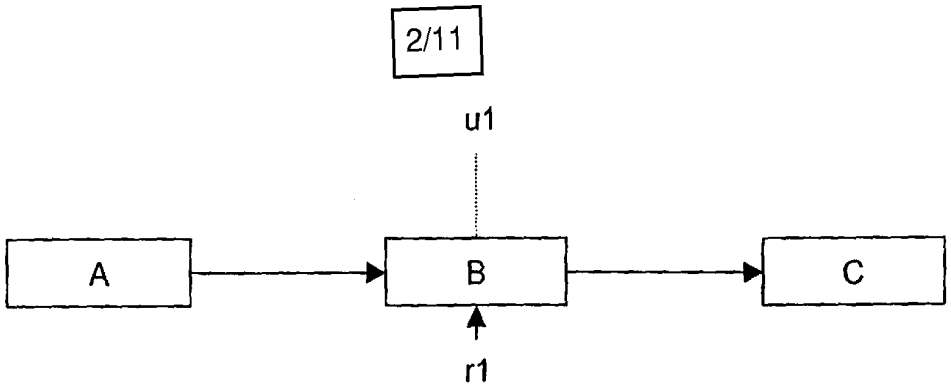


FIG. 2A (PRIOR ART)

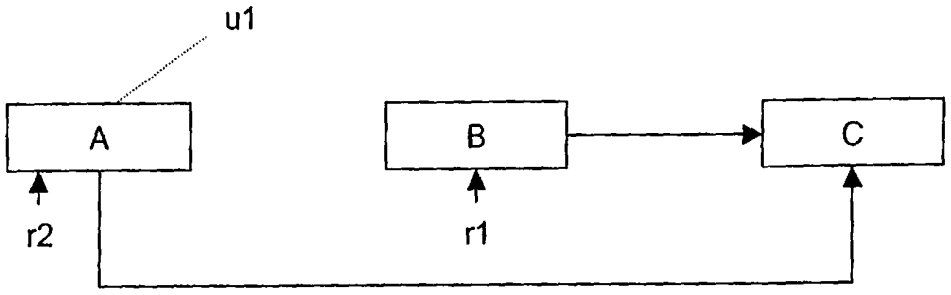


FIG. 2B (PRIOR ART)

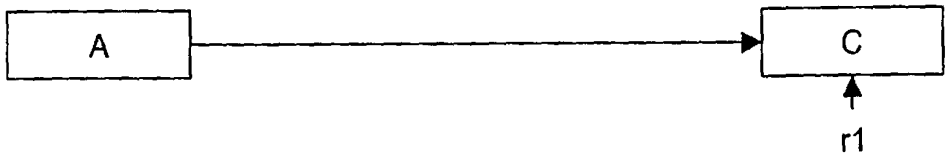


FIG. 2C (PRIOR ART)

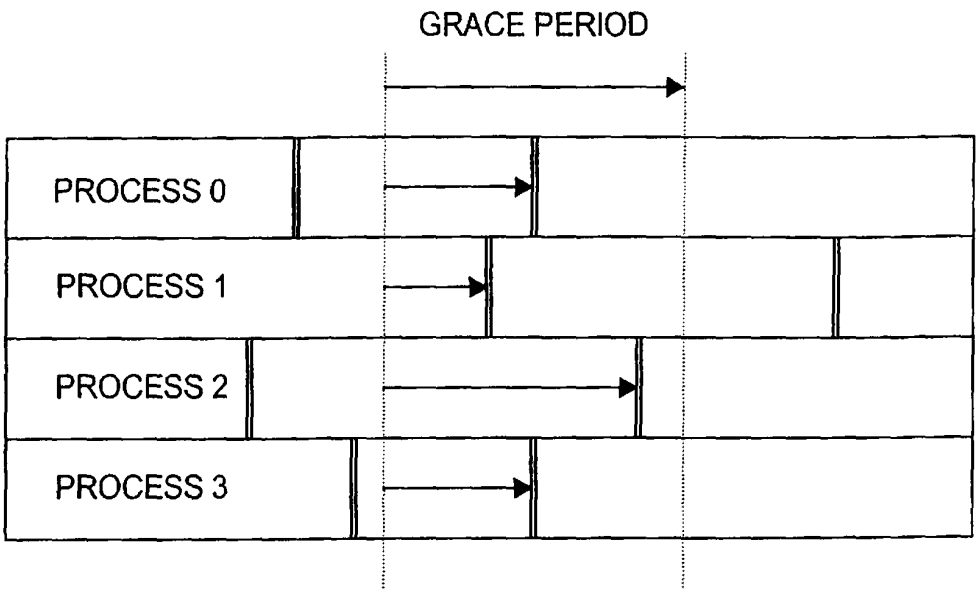


FIG. 3 (PRIOR ART)

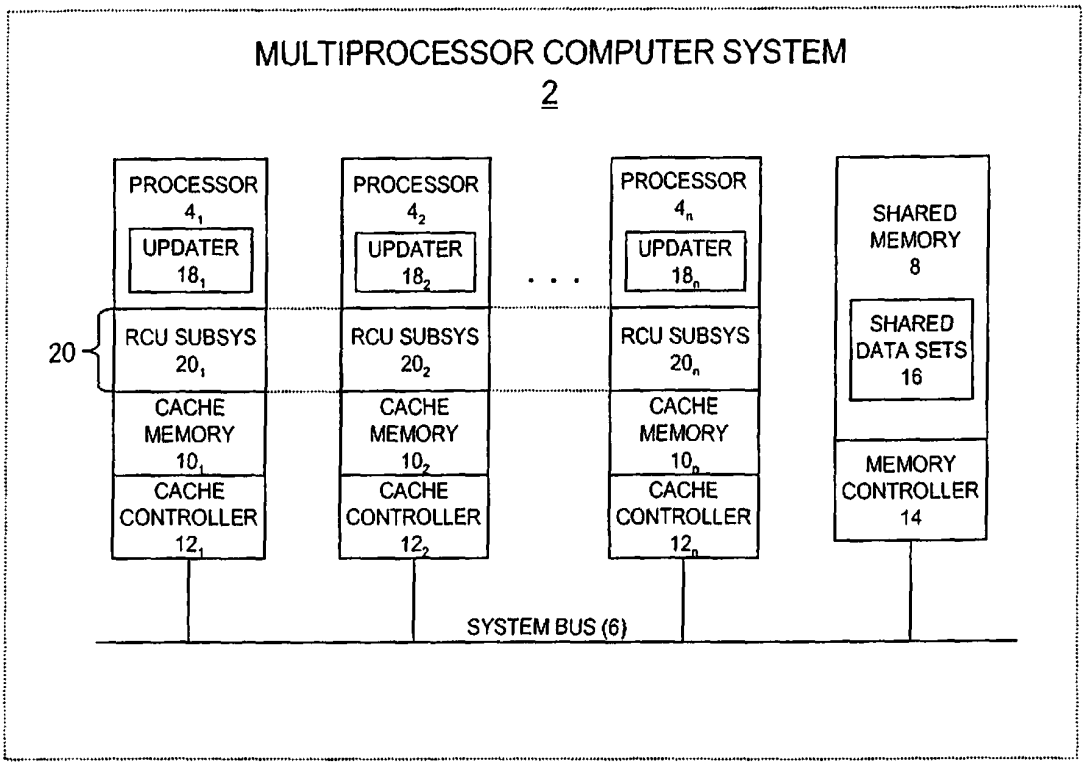


FIG. 4

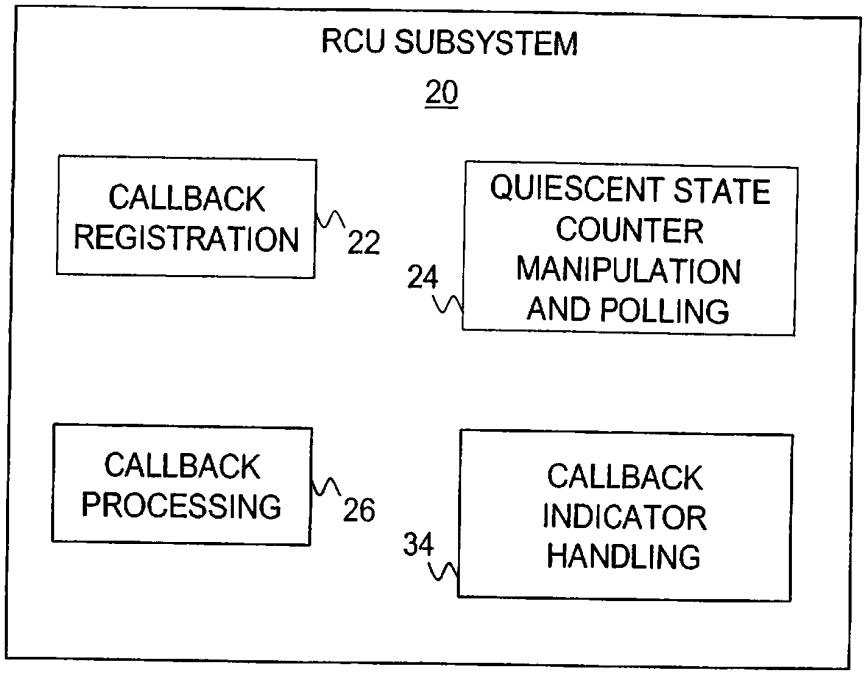

RCU SUBSYSTEM
20
CALLBACK REGISTRATION
22
QUIESCENT STATE COUNTER MANIPULATION AND POLLING
24
CALLBACK PROCESSING
26
CALLBACK INDICATOR HANDLING
34

FIG. 5

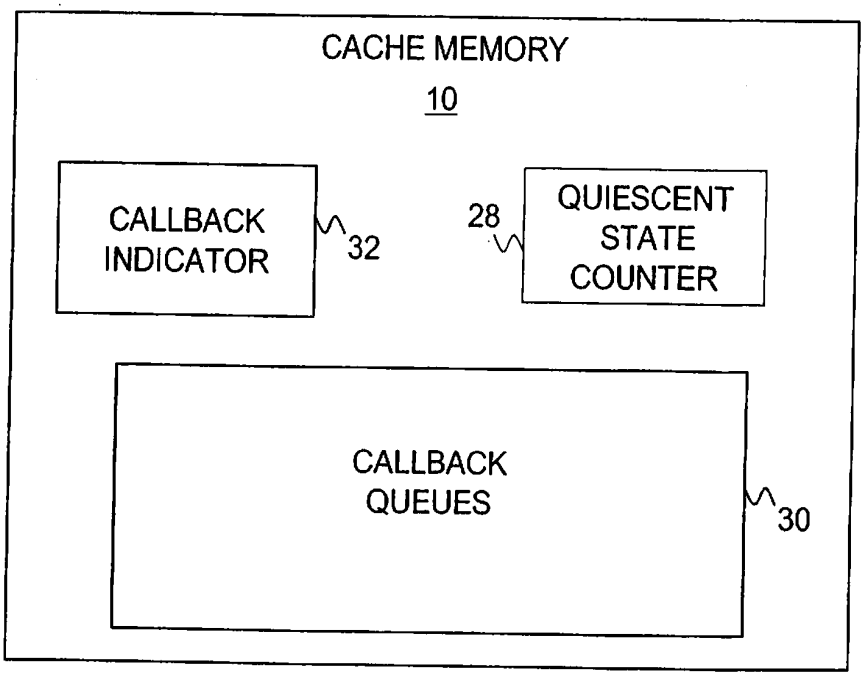

CACHE MEMORY
10
CALLBACK INDICATOR
32
28
QUIESCENT STATE COUNTER
CALLBACK QUEUES
30

FIG. 6

	VALUES AT TIMESTAMPS								
	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8
PROC 0	4	4	4	4	8	8	8	8	12
PROC 1	3	3	3	7	7	7	7	11	11
PROC 2	2	2	6	6	6	6	10	10	10
PROC 3	1	5	5	5	5	9	9	9	9

FIG. 7 (PRIOR ART)

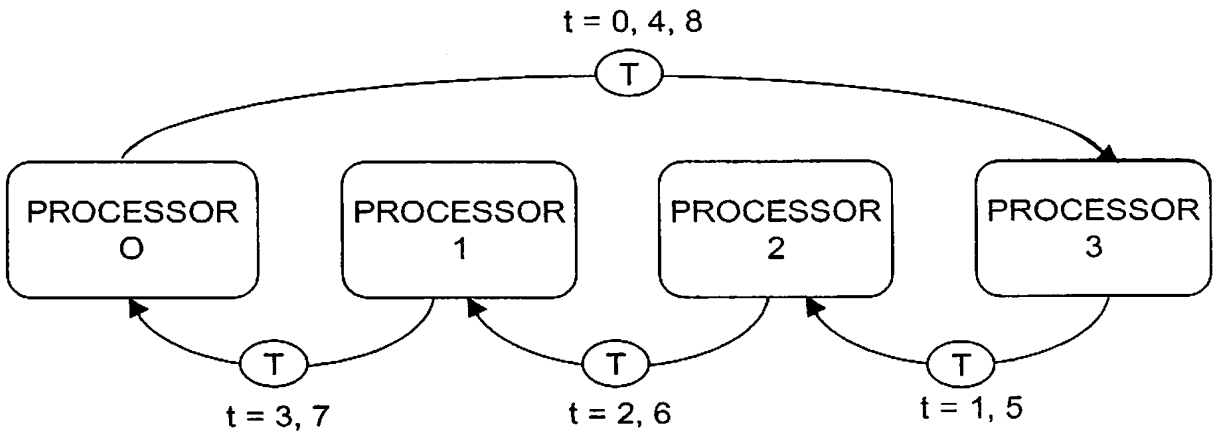


FIG. 8 (PRIOR ART)

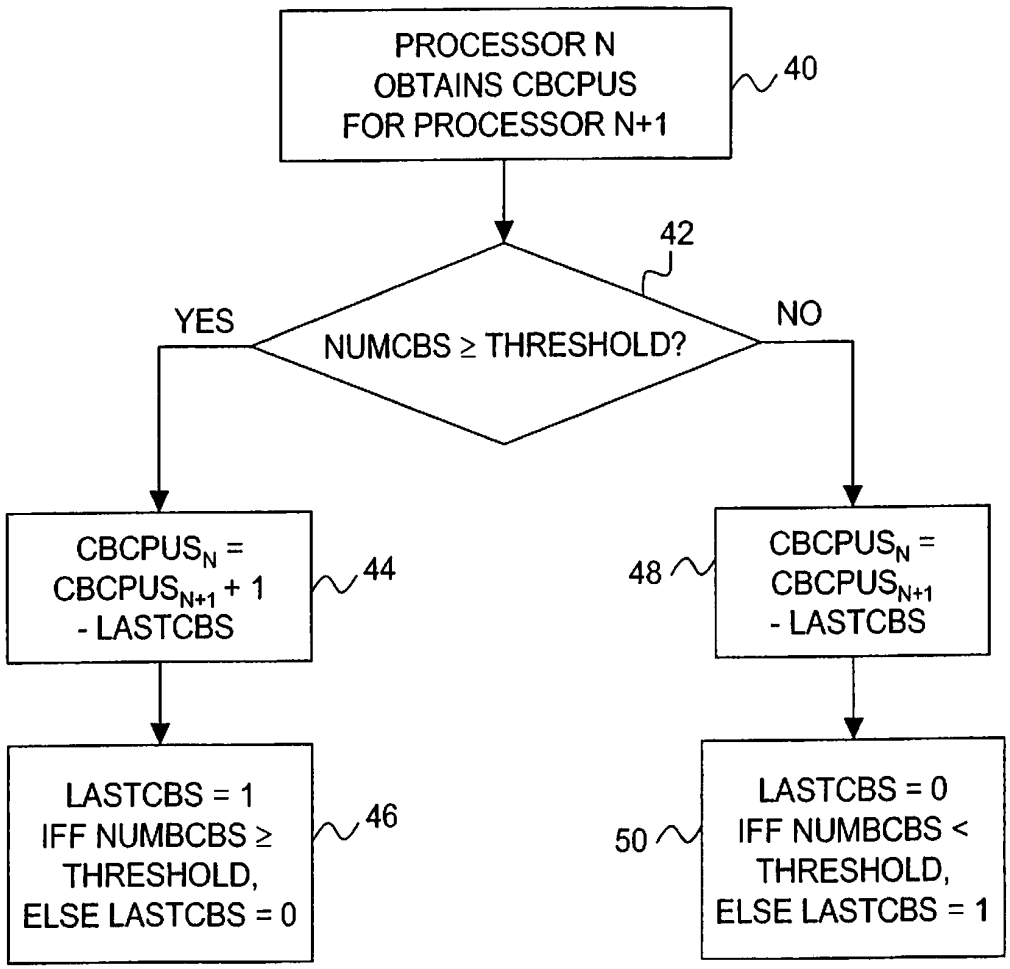

PROCESSOR N OBTAINS CBCPUS FOR PROCESSOR N+1
40
42
NUMCBS ≥ THRESHOLD?
YES
NO
CBCPUS$_N$ = CBCPUS$_{N+1}$ + 1 - LASTCBS
44
LASTCBS = 1 IFF NUMBCBS ≥ THRESHOLD, ELSE LASTCBS = 0
46
48
CBCPUS$_N$ = CBCPUS$_{N+1}$ - LASTCBS
LASTCBS = 0 IFF NUMBCBS < THRESHOLD, ELSE LASTCBS = 1
50

FIG. 9

	VALUES AT TIMESTAMPS								
	t=0	t=1	t=2	t=3	t=4	t=5	t=6	t=7	t=8
PROC 0	4 / 0	4 / 0	4 / 0	4 / 1	4 / 1	4 / 1	4 / 1	8 / 1	8 / 0
PROC 1	3 / 0	3 / 0	3 / 1	3 / 1	3 / 1	3 / 1	7 / 1	7 / 0	7 / 0
PROC 2	2 / 0	2 / 1	2 / 1	2 / 1	2 / 1	6 / 1	6 / 0	6 / 0	6 / 0
PROC 3	1 / 0	1 / 0	1 / 0	1 / 0	5 / 1	5 / 1	5 / 1	5 / 1	9 / 1

FIG. 10

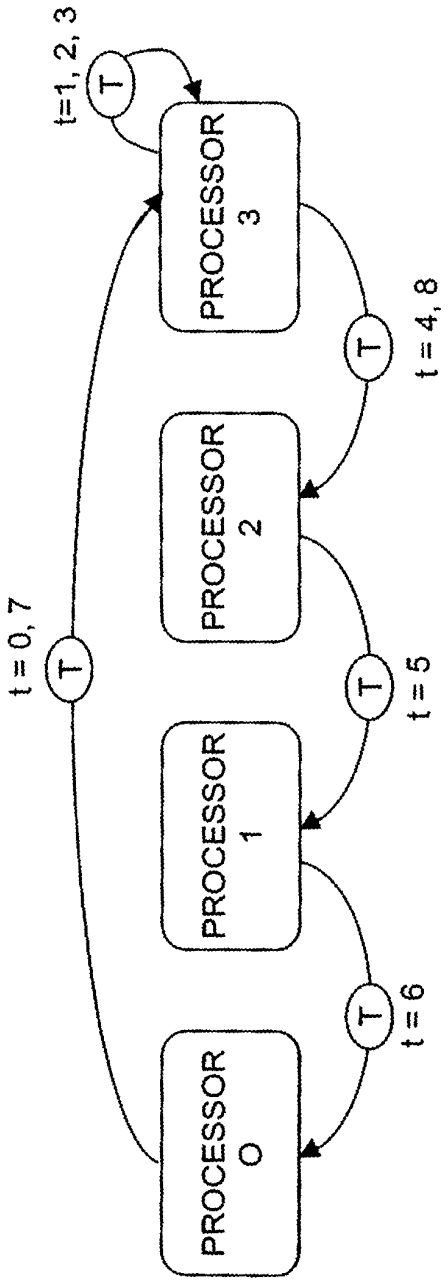


FIG. 11

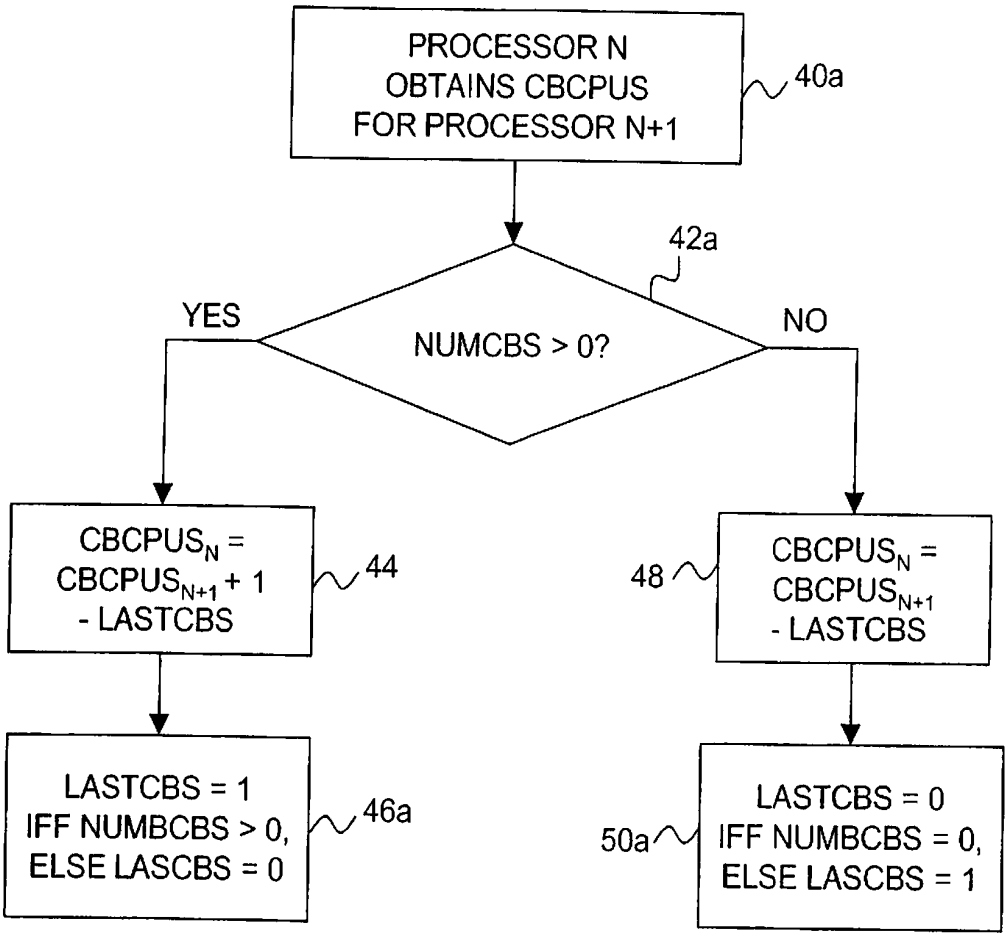

PROCESSOR N
OBTAINS CBCPUS
FOR PROCESSOR N+1
40a
42a
NUMCBS > 0?
YES
NO
CBCPUS$_N$ =
CBCPUS$_{N+1}$ + 1
- LASTCBS
44
LASTCBS = 1
IFF NUMBCBS > 0,
ELSE LASCBS = 0
46a
48
CBCPUS$_N$ =
CBCPUS$_{N+1}$
- LASTCBS
50a
LASTCBS = 0
IFF NUMBCBS = 0,
ELSE LASCBS = 1

FIG. 12

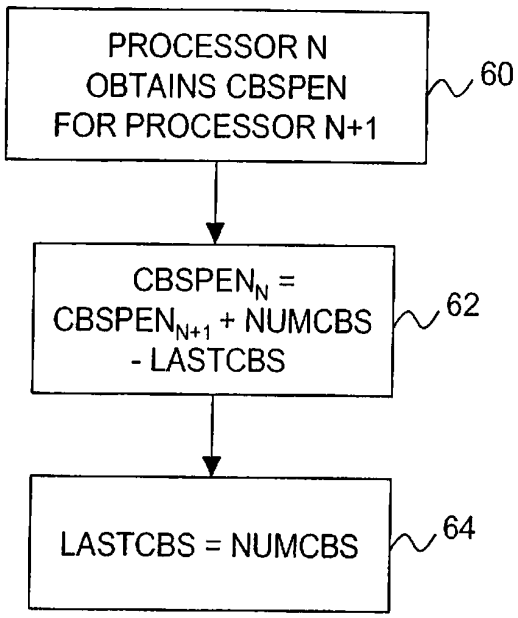

PROCESSOR N
OBTAINS CBSPEN
FOR PROCESSOR N+1
60
CBSPEN$_N$ =
CBSPEN$_{N+1}$ + NUMCBS
- LASTCBS
62
LASTCBS = NUMCBS
64

FIG. 13

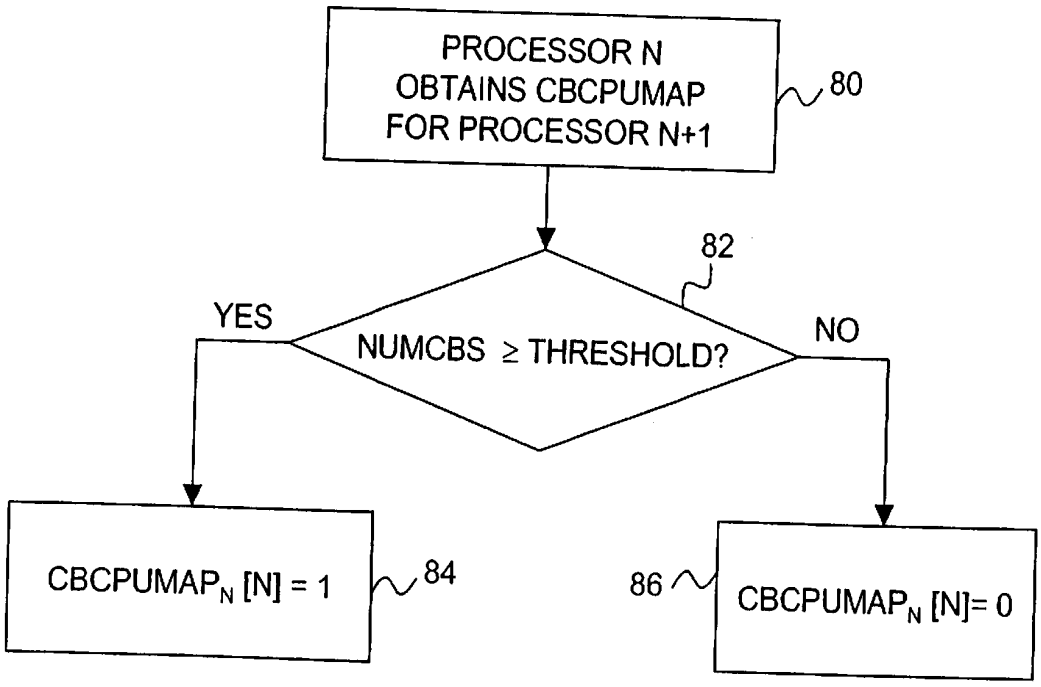

PROCESSOR N
OBTAINS CBCPUMAP
FOR PROCESSOR N+1
80
82
YES
NUMCBS ≥ THRESHOLD?
NO
CBCPUMAP$_N$ [N] = 1
84
86
CBCPUMAP$_N$ [N]= 0

FIG. 14

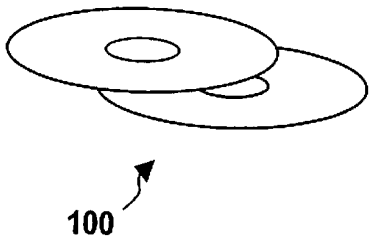


FIG. 15

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2005/055240

A. CLASSIFICATION OF SUBJECT MATTER
G06F9/46

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	PAUL E. MCKENNEY ET AL: "Read Copy Update" OTTAWA LINUX SYMPOSIUM 2002, [Online] 29 June 2002 (2002-06-29), pages 338-367, XP002372599 Retrieved from the Internet: URL:http://www.linux.org.uk/{ajh/ols2002_proceedings.pdf.gz> [retrieved on 2006-03-16] cited in the application page 339 - page 343	12
A	the whole document	1-11, 13-15,30

-/--

[X] Further documents are listed in the continuation of Box C.

[ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
17 March 2006	30/03/2006

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Dewyn, T

INTERNATIONAL SEARCH REPORT

International application No
PCT/EP2005/055240

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	PAUL E. MCKENNEY ET AL: "Read-Copy Update"[Online] July 2002 (2002-07), XP002372502 Retrieved from the Internet: URL:http://www.rdrop.com/users/paulmck/rclock/rcu.2002.07.08.pdf> [retrieved on 2006-03-15] page 2	12
A	the whole document	1-11, 13-15,30
A	ARCANGELI A ET AL: "Using read-copy-update techniques for System V IPC in the Linux 2.5 kernel" FREENIX TRACK 2003 USENIX ANNUAL TECHNICAL CONFERENCE. PROCEEDINGS USENIX ASSOC BERKELEY, CA, USA, 2003, pages 297-309, XP002372501 ISBN: 1-931971-11-0 the whole document	1-15,30
A	GAMSA B ET AL: "Tornado: maximizing locality and concurrency in a shared memory multiprocessor operating system" OPERATING SYSTEMS REVIEW ACM USA, 1998, pages 87-100, XP002372503 ISSN: 0163-5980 the whole document	1-15,30