METHOD OF ASYNCHRONOUS MANAGEMENT OF A CIRCULAR QUEUE IN A MULTIPROCESS ENVIRONMENT

A circular queue is asynchronously accessed and managed by two separate processing elements. Each data element is placed on the queue together with a zero data element that both marks the tail of the queue and signifies that the queue is empty. Data elements are removed from the queue in the order in which they were stored (first-in–first-out) and a manner that allows multiple, concurrent access to the queue. When the queue is accessed to remove a data element the element is first tested. If it is non-zero, the removal process continues; if zero, the queue is considered empty. The management of the queue permits dynamic re-sizing the queue (i.e., making the queue larger or smaller) while data elements are being added and/or removed.
FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Albania</td>
<td>ES</td>
<td>Spain</td>
<td>LS</td>
<td>Lesotho</td>
</tr>
<tr>
<td>AM</td>
<td>Armenia</td>
<td>FI</td>
<td>Finland</td>
<td>LT</td>
<td>Lithuania</td>
</tr>
<tr>
<td>AT</td>
<td>Austria</td>
<td>FR</td>
<td>France</td>
<td>LU</td>
<td>Luxembourg</td>
</tr>
<tr>
<td>AU</td>
<td>Australia</td>
<td>GB</td>
<td>United Kingdom</td>
<td>LV</td>
<td>Latvia</td>
</tr>
<tr>
<td>AZ</td>
<td>Azerbaijan</td>
<td>GE</td>
<td>Georgia</td>
<td>MC</td>
<td>Monaco</td>
</tr>
<tr>
<td>BA</td>
<td>Bosnia and Herzegovina</td>
<td></td>
<td>GE</td>
<td>Georgia</td>
<td>MD</td>
</tr>
<tr>
<td>BB</td>
<td>Barbados</td>
<td>GH</td>
<td>Ghana</td>
<td>MG</td>
<td>Madagascar</td>
</tr>
<tr>
<td>BE</td>
<td>Belgium</td>
<td>GN</td>
<td>Guinea</td>
<td>MK</td>
<td>The former Yugoslav Republic of Macedonia</td>
</tr>
<tr>
<td>BF</td>
<td>Burkina Faso</td>
<td>GR</td>
<td>Greece</td>
<td>ML</td>
<td>Mali</td>
</tr>
<tr>
<td>BG</td>
<td>Bulgaria</td>
<td>HU</td>
<td>Hungary</td>
<td>MN</td>
<td>Mongolia</td>
</tr>
<tr>
<td>BJ</td>
<td>Benin</td>
<td>IE</td>
<td>Ireland</td>
<td>MR</td>
<td>Mauritania</td>
</tr>
<tr>
<td>BR</td>
<td>Brazil</td>
<td>IL</td>
<td>Israel</td>
<td>MW</td>
<td>Malawi</td>
</tr>
<tr>
<td>BY</td>
<td>Belarus</td>
<td>IS</td>
<td>Iceland</td>
<td>MX</td>
<td>Mexico</td>
</tr>
<tr>
<td>CA</td>
<td>Canada</td>
<td>IT</td>
<td>Italy</td>
<td>NE</td>
<td>Niger</td>
</tr>
<tr>
<td>CF</td>
<td>Central African Republic</td>
<td></td>
<td>JP</td>
<td>Japan</td>
<td>NL</td>
</tr>
<tr>
<td>CG</td>
<td>Congo</td>
<td>KE</td>
<td>Kenya</td>
<td>NO</td>
<td>Norway</td>
</tr>
<tr>
<td>CH</td>
<td>Switzerland</td>
<td>KG</td>
<td>Kyrgyzstan</td>
<td>NZ</td>
<td>New Zealand</td>
</tr>
<tr>
<td>CI</td>
<td>Côte d'Ivoire</td>
<td>KP</td>
<td>Democratic People's Republic of Korea</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td>Cameroon</td>
<td>KR</td>
<td>Republic of Korea</td>
<td>PL</td>
<td>Poland</td>
</tr>
<tr>
<td>CN</td>
<td>China</td>
<td>KZ</td>
<td>Kazakhstan</td>
<td>PT</td>
<td>Portugal</td>
</tr>
<tr>
<td>CU</td>
<td>Cuba</td>
<td>LC</td>
<td>Saint Lucia</td>
<td>RO</td>
<td>Romania</td>
</tr>
<tr>
<td>CZ</td>
<td>Czech Republic</td>
<td>LI</td>
<td>Liechtenstein</td>
<td>RU</td>
<td>Russian Federation</td>
</tr>
<tr>
<td>DE</td>
<td>Germany</td>
<td>LK</td>
<td>Sri Lanka</td>
<td>SD</td>
<td>Sudan</td>
</tr>
<tr>
<td>DK</td>
<td>Denmark</td>
<td>LR</td>
<td>Liberia</td>
<td>SE</td>
<td>Sweden</td>
</tr>
<tr>
<td>EE</td>
<td>Estonia</td>
<td></td>
<td></td>
<td>SG</td>
<td>Singapore</td>
</tr>
<tr>
<td>SI</td>
<td>Slovenia</td>
<td>SK</td>
<td>Slovakia</td>
<td>SN</td>
<td>Senegal</td>
</tr>
<tr>
<td>SZ</td>
<td>Swaziland</td>
<td></td>
<td></td>
<td>SZ</td>
<td>Swaziland</td>
</tr>
<tr>
<td>TD</td>
<td>Chad</td>
<td></td>
<td></td>
<td>TG</td>
<td>Togo</td>
</tr>
<tr>
<td>TJ</td>
<td>Tajikistan</td>
<td></td>
<td></td>
<td>TR</td>
<td>Turkmenistan</td>
</tr>
<tr>
<td>TT</td>
<td>Trinidad and Tobago</td>
<td></td>
<td></td>
<td>UK</td>
<td>Ukraine</td>
</tr>
<tr>
<td>UA</td>
<td>Uganda</td>
<td></td>
<td></td>
<td>US</td>
<td>United States of America</td>
</tr>
<tr>
<td>VN</td>
<td>Viet Nam</td>
<td></td>
<td></td>
<td>YU</td>
<td>Yugoslavia</td>
</tr>
<tr>
<td>ZW</td>
<td>Zimbabwe</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
METHOD OF ASYNCHRONOUSLY MANAGEMENT OF A CIRCULAR QUEUE IN A MULTIPROCESS ENVIRONMENT

BACKGROUND OF THE INVENTION

The present invention relates generally to computing systems and in particular to a method of managing an event queue or buffer that is asynchronously accessed and managed by at least two different, unsynchronized elements.

In data processing activity it is often necessary to temporarily store data items in time-ordered fashion so that those earlier stored can be retrieved first. One method of such storage is a queue: a data structure that stores data elements that are usually retrieved in the same order they are received, i.e., a data structure that operates in accordance with a first-in-first-out (FIFO) constraint. (Other types of queues may base retrieval of data items on factors other than when stored.)

While queue operation, use, and management is relatively simple in the context of a single application/processor, queue management can become more difficult in operating environments in which there exist multiple processing units and/or processes needing access to a single queue. Typically, to ensure that the queue is not changed by a second processing unit or process while being accessed by a first processing unit or process, a lock will be established by the first processing unit/process to preclude access while it is changing or accessing the queue. In a multiple process environment (e.g., one a multiple thread system) this method of ensuring queue consistency can seriously affect system efficiency. This is an especially acute problem when the threads are running as a user mode process. For example, suppose a thread seeks to access a queue, locking it to ensure it is not changed before it is through. But, before the thread can complete its use of the queue, it is preempted (i.e., its time slice has expired), and it must wait until it can again operate. Until then, the queue cannot be accessed by any other thread, process or processor.
Also, suppose the queue is used by two processing elements, one to place data elements on the queue, and another to retrieve those data elements. The queue pointers, the address information that identifies where in the queue the next data item is to be stored, or the location of the data item next in line for retrieval, must be separately managed.

Typically, one unit, for example the unit removing or pulling data from the queue, must read (or have written to it) the queue pointer of the other unit adding or pushing data onto the queue so that the two pointers can be compared to determine if the queue is empty or contains data to be pulled (removed). This separate read or write has a negative impact on performance.

SUMMARY OF THE INVENTION

The present invention provides a method of constructing and using a queue for storing data elements so that it can be asynchronously managed by separate processing elements, one to sequentially place data elements on the queue, the other to sequentially remove them from the queue in the order they were placed on the queue. In addition, the method of removing the data elements is performed in a manner that allows multiple processes (e.g., multiple processes within running on one processor, or multiple processors as in a symmetric multiple processing (SMP) system, as well as multiple concurrent threads of either) to access the queue for removing data elements without conflict, and in a manner that does not unduly impede operation of the queue.

The invention is developed to avoid problems (some of which are described above) encountered when using a queue in a multiprocess environment such as may be found in a SMP system (e.g., a 4-way SMP system, comprising four processor units). An SMP system can have one or more processes concurrently running on the processor units of the system, and well as multiple threads of those processes. The invention permits data elements to be pulled ("removed") from the queue by the concurrent processes/threads without the inefficiency of software locks.
According to a broad aspect of the invention, a multiple-entry queue is formed in a circular configuration, and maintained in memory. A head pointer resides in shared memory that is used to access the entry containing the data element next to be removed - if any.

Another processing element keeps a tail pointer that is indicative of the entry location whereat the next data element is to be stored. Each data element is stored together with a zero entry that marks the tail of the queue. Data elements are removed by first checking to determine if the content of the queue entry identified by the head pointer is non-zero. If so, the data element is removed by an atomic operation that uses a test and conditional store operation on the shared (by the multiple processes or threads) head pointer. First, the head pointer is checked to determine that it has not changed, i.e., that it is the same as that used to retrieve the data element. If so, the head pointer is modified to identify the entry containing the next data element to be retrieved.

The head pointer comprises an offset value and a "wrapcount" value. The offset value is combined with a base value, identifying where in memory the queue resides, to locate the head entry, i.e., the entry of the queue containing the oldest data element that is the next to be retrieved. When retrieving a data element, only the offset value of the head pointer is used (with the base value) to develop the memory address of the queue entry. However, the test and conditional store operation uses the entire head pointer, including the wrapcount (which changes each time the removal of data elements wraps around). The wrapcount is used to signal to a thread that may be pre-empted after it has read the head of the queue, but before it can perform the test and conditional store operation, that a wrap has occurred. That is, while the thread was "asleep" the addition and removal (by other threads) of data elements to and from the queue wraps around so that when the pre-empted thread is brought back, the offset value is the same as that used by the thread before pre-emption. However, the wrapcount value will have changed. Thus, the test of the head pointer will fail, and the thread will discard the data element earlier accessed. Without this feature, and under the conditions described, the awakened thread would complete the test and conditional store
operation, leaving a head pointer that is indicative of the data element one beyond that next for removal.

The configuration of the queue, and the management of its pointers, according to the present invention, permits the queue to be dynamically re-sized (replaced with a larger or smaller queue) with a minimum of effort. The re-size operation can take place while data elements are being added to the queue and processes are attempting to remove older data elements from the queue. Furthermore, no software locks are required to prevent these processes from accessing the queue during the re-size operation. Broadly the steps to re-size the queue to one that is larger for example, include the steps of:

1. Allocate the physically contiguous memory for the new queue. Initialize the new queue by zeroing all the entries.
2. Temporarily let Users of the queue think it is empty by setting the Head pointer to point to a zero entry. (This will prevent access because the users will see the non-zero data element.)
3. Update the queue base address in memory, the queue size and the head pointer.
4. Update the tail pointer (used by the second processing unit) so it points to the new queue.
5. Move any remaining data elements from the old queue to the new queue.
6. If data elements are on the new queue, unblock any users waiting on the queue and let them access the new head pointer.

The invention finds particular advantage in multitasking environments due to its ability to provide concurrent access to the queue for removing data elements without the use of software locks or other means to prevent other user processes (e.g., threads) from accessing the queue when in use. That is, multiple threads or processes can pull data elements from the queue in a manner that does not require a software lock mechanism between them. This can be important given the overhead often imposed by such locking (or other) techniques. Thus, the present invention allows for efficient operation with multiple concurrent threads running to simultaneously
pull items from the queue. As will be seen, the present invention also takes into account the possibility of a thread running in user mode being put to "sleep" in the middle of accessing the queue long enough so that it wraps around to the same spot. (As used herein, "user mode" is meant to refer to the processor operations performed as a result of execution of user application programs or processes, or threads of those application programs or processes; "kernel mode" is meant to refer to processor operations performed as a result of the underlying operating system running on the processor.)

Although preemption during access is provided for and even allowed, the re-sizing operation, which is done in kernel mode, is preferably performed (for performance reasons) so that steps 2 through 5 are executed without being preempted.

These and other aspects and advantages of the invention will become evident to those skilled in this art upon a reading of the following written description of the invention, which should be taken in conjunction with the accompanying drawings.

Further, data elements can be located and removed from the queue without the requirement of reading a register to determine if a new data element has arrived on the queue. In addition,

**BRIEF DESCRIPTION OF THE DRAWINGS**

Fig. 1 is simplified block diagram illustration of computing system architecture employing the present invention to manage an event queue kept in main memory;

Figs. 2A and 2B respectively illustrate the head and tail pointer data structures used to locate the entry locations of the event queue whereat the data element next to be retrieved is stored and the location whereat the next data element is to be stored;

Figs. 3A, 3B, and 3C diagrammatically illustrated use of an event queue according to the present invention;
Fig. 4 illustrates the procedure used to form the head pointer used to locate the data element in the queue next to be retrieved;

Fig. 5 illustrates generally the steps taken to retrieve a data item from the queue; and

Figs. 6A - 6G diagrammatically illustrate the steps taken to dynamically re-size an established, used queue according to the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT

As indicated, the present invention finds particular advantage in a multiple process environment - either a multitasking or multiprocessing environments. Thus, Fig. 1 illustrates, for example, a multiple processor node 10 that forms a part of a larger system that may be configured as a multiple processor cluster. The node 10 would have, for example, multiple processors. Although only one processor unit 12 is actually shown in Fig. 1, in order not to over complicate the discussion, it will be appreciated by the reader that if the node 10 is configured as a symmetrical processing system (SMP), processor unit 12 would, in fact, be multiple processors, and will be so referenced in the remainder of this disclosure. Continuing with Fig. 1, the processor(s) 12 is(are) connected to a shared memory 14 via a host bus 16. The node 10 may communicatively connect to the remainder of the larger system of cluster (not shown) through an interface unit 20 which handles transmission and reception tasks for the node 10. The interface unit connects to an internode bus system 22 (which may be multiple buses) that, in turn, connect to the other nodes (not shown) of the cluster. The interface unit 20 is preferably a state machine based design, although it could be microprocessor-based if speed and efficiency are not all that important.

The interface unit 20 also connects to the processor(s)/memory 12/14 via a separate bus 24 (which could be a peripheral component interconnect - PCI - bus) and a (PCI) bridge 26. Typically, the interface unit 20 would have direct memory access (DMA) to the memory 14, according to standard PCI bus protocol.
The present invention provides a storage facility in the form of a data structure where data elements can be stored in time-ordered fashion, e.g., when received. For example, processes running on the processor(s) 12 may need to know when an initiated communication is sent to or received from the larger system by way of the interface unit in order to perform activities related to the communication. Or, the processes may need to know of other events that occur involving the interface unit. The present invention, then provides a data structure, in the form of a event queue 30, for containing data elements describing events as they occur, and providing an indication of what activity needs performing, either in the form of a pointer to additional information for handling the related matter, or the data element may be self-contained, i.e., provide all the information needed to perform whatever activity needs performing relating to the event the data element describes.

The event queue 30 is a circular buffer of 4 byte entries 32 (32a, 32b, ..., 32i, ..., 32m), and is structured to have space for \(2^n\) entries, where \(n\) ranges from 1 to 16, although there is no harm in a larger event queue than was requested by the user. As is typical, data elements are removed from the event queue 30 in the order they were inserted, i.e., in FIFO fashion. A head pointer value 36 is kept in memory 14 where it is accessible to the processes running on the processor(s) 12. (The event queue and head pointer reside in the memory space of the process that owns, i.e., created, the event queue.) The head pointer value provides information as to the location of the entry containing the earliest-stored data element. Tail pointer information is held in a tail pointer register 40 that is preferably maintained in a memory 42 local to the interface unit 20. The content of the tail pointer register 40 points to the event queue entry (32i in Fig. 1) of the event queue 30 at which the next data element added will be stored.

The interface unit 20 adds event queue entries (QE) 32 to the tail, and processes running on the processor(s) 12 remove QEs from the head. The interface unit 20 adds items to decreasing addresses in the queue 30, that is the queue grows to lower memory.
The content of the tail register 40, the tail pointer value, always points to an entry 32 of the queue 30 that contains a zero (e.g., entry 32i in Fig. 1). The interface unit 20 writes this zero entry when the it adds a new data element to the event queue 30, and decrements the tail pointer value. If the event queue is empty, the head and tail values will point to the same zero entry. Of course, using a zero entry to identify the tail of the queue will consume one of the 2n entries in the event queue 30. Therefore, only 2^n-1 entries in an event queue are usable.

Digressing for the moment, the zero data element performs an important function: to provide an indication of whether or not the queue is empty. Prior queue management techniques would compare the head pointer and tail pointer values to one another: if they were equal, the queue was considered empty; if not, the queue was considered not empty. But, this approach would require that either the interface unit 20 write the head pointer value to the memory 14, or the processor(s) 12 would read the head pointer register 40, to make the tail pointer value available for the comparison. This, however, would be inefficient and costly (in terms of time). Instead, according to the present invention, writing a zero data element one beyond each data element stored by the interface unit 30 is, when retrieved and examined, used to denote that the queue was empty. Thus, the zero data element identifies to the removal process, when tested, that the queue is empty. And, it is better that the entity adding data elements to the queue write this "empty" flag. If the processes or threads removing data elements were used to write the zero, pre-emption problems discussed above would also operate here to cause problems (i.e., a thread is pre-empted before it can store a zero might result in a subsequent thread removing a stale date element).

As shown, there is no virtual to physical address translation logic, and the interface unit 20 requires that the event queue 30 be physically contiguous. The event queue start address need only to be 4 byte aligned. While the event queue must have 2n entries, there is no requirement that the event queue be aligned to a 2n boundary.
Preferably, the event queue 30 is physically contiguous and appear in the user-mode process's virtual address space. Thus, the memory for the event queue 30 must be allocated to be physically contiguous, and then mapped into virtual space for a client-process. A single 4KB page can hold 1024 event queue entries. Any larger event queue requires two or more physically contiguous pages.

The tail pointer information resides in the interface unit 20 (i.e., tail pointer register 40), and is preferably never read from or written to by a user agent. Rather, it is believed better that an operating system (Kernel agent) be designated to write the tail pointer information to the tail pointer register 40 only when creating the event queue 30 or re-sizing the event queue (discussed further below). The tail pointer can be initialized to anywhere in the event queue 30. In the example shown in Figs. 3A-3C, discussed below, the head pointer and tail pointer were initially set to Base + 2^{Qsize+3} - 4, the highest addressed entry in the event queue.

When the event queue 30 is created, there are several items written to the tail pointer register 40 to describe the location in memory of the tail of the event queue 30, including:

Tail Offset: The offset, in number of entries of the event queue 30, from the base to the current (empty, or zero) tail location.

Queue Size. The total number of entries in a CQ is 2^{Qsize+1} An entry containing all zeros is used to mark the tail of the queue so the number of usable entries is 2^{Qsize+1} - 1.

Queue Base Address: The address of the lowest CQ entry in the queue.

Fig. 2A shows the head pointer information. A 64-bit data word 50 contains the head offset 52 and wrapcount 54 values. In addition to this is the base address 56, QBase, of the location of the event queue 30 in memory 14, and a queue size value 58 (QSize) that is indicative of the size of the queue. The tail pointer information is shown in Fig. 2B as a 64-bit data word 60 (kept in the tail pointer register 40 - Fig. 1), containing the location in memory of the event queue 30, QBase 62, and the offset 64 into the event queue to the
entry containing a zero data element to demark the tail of the queue.
The interface unit 20 also has a queue size value 68 (QSize),
describing the size of the queue.

Referring now to Figs. 3A-3C, an example of the use and
construction of the event queue 30 is illustrated. Referring first to
Fig. 3A, the QBase value is shown as pointing to the location in the
memory 14 (Fig. 1) at which the bottom or base of the event queue 30
resides. The head and tail pointer values contain offsets into the
event queue 30 respectively the entry 32 of the queue 30 whereat the
next data element to be removed is located, and the tail of the queue.
The size of the queue is 8 entries. Thus, a size value (QSize) 66 is
maintained for reasons that will be explained below, and identifies
actual size of the event queue 30, in terms of the number of entries.
The size value, plus 1, forms a power of 2 to define the number of
queue entries. Thus, with a size value of 2, the event queue is
defined to have 8 ($2^{2+1}$) entries.)

Continuing, Fig. 3A shows an empty event queue 30 having eight
entries. Thus, the head pointer information 50a available to the
processes running on the processor(s) 12 and that contained in the
tail pointer register 40 of the interface unit 20, will point to the
same (empty) entry. Now, assume that the interface unit 20 adds three
data elements, QEO, QE1, and QE2, to the event queue 30. The head
pointer information 50a will point to the first data element, QEO,
inserted in the queue 30 by the interface unit 20, and will continue
to do so until that data element is removed from the queue (Fig. 3B).
The tail pointer information now points to the entry one past the last
data element (QE2) inserted by the interface unit 20. Before each
data item is inserted by the interface unit 20, a zero data element is
written to the entry that immediately follows. Thus, when the data
element QE2 was written to the entry location 14 (hexadecimal), a zero
data element was written to the entry location 10 (hexadecimal). This
is done as a single 8 byte block write operation on the PCT bus. A
PCI block write starts at the lowest addressed 4 bytes and writes to
the highest 4 bytes. Therefore, pci block write operation always
writes the zero data element before writing the new data element to
the event queue 30.
Fig. 3C shows the event queue 30 after the interface unit 20 has inserted eleven more data elements, and eleven data elements have been removed from the queue by processes (threads) running on the processor(s) 12. Note that the tail pointer information is still left pointing to a queue entry (at location 04) that contains a zero value.

The interface unit 20 is preferably configured to perform only a single PCI block write for each data element written to the event queue 30, and should not be required to do a separate write to the shared memory 14 for the tail pointer. In addition, if a PCI bus is used to connect the interface unit 20 to the processor(s)/memory 12/14, note should be taken of the fact that, compared to PCI write operations, reads across the PCI bus 24 are relatively slow. The interface unit 20 should not have to do a PCI read of the memory 14 in order to manage the event queue 30. Similarly, a process running on the processor(s) 12 should not have to perform a PCI read of an interface unit 20 resident register (e.g., tail pointer register 40) in order to manage the event queue 30. Accordingly, the processor(s) 12 (i.e., processes) should not have to read the tail pointer register 40 in order to determine if an item is on the queue. And, concurrent threads should be able to remove items from a event queue 30 without employing any kind of locking mechanisms.

As indicated above (with respect to Fig. 2A), the head pointer information is not a simple pointer to the current head data element in the event queue 30. Instead, the head pointer information is actually a 64 bit (aligned to an 8 byte boundary) combination of the wrapcount 52 and an offset value 54. The offset value 54 of the head pointer information is added to the event queue base value 56 to produce the address of the desired event queue entry. Since the event queue can be of variable size, a mask computed from the QSize value is needed to separate the wrapcount 52 field from the offset 54 field. Shown in Fig. 4 is a diagram illustrates this operation.

As Fig. 4 shows, the 64-bit word, containing the wrapcount and offset 52, 54, is ANDed with a value that is based upon the size of the event queue 30 to that only the offset value 50 is passed. The offset is then left-shifted 2 bit positions (to account for the fact
that each entry is four bytes), and added to the value QBase. The result is the memory location of the entry that forms the head of the event queue 30.

The memory location of the queue containing the zero data element, marking the tail of the queue (and one entry just beyond the last data element added) is formed in the same manner, i.e., according to the procedure shown in Fig. 4, using the tail pointer information 60 (QBase 62 and offset 64 - Fig. 2B) and QSize 68. (Of course, it will be evident that the QBase and QSize values 52/66, 58/68 used to develop the entry addresses for the head and tail are the same.)

The actual value of the wrapcount value is not important for address calculations. It is only used by the operation that updates the head pointer as discussed further below. The wrapcount value 52 ensures that a stale offset value 54 does not mistakenly match an offset that has already wrapped around. The event queue head pointer wrapcount value is preferably initialized with a random number.

The event queue 30 grows to low memory. To advance the pointer one item, the entire 64-bit head pointer word 50 is decremented by one. This will correctly advance the wrapcount value 52 when the event 30 wraps since decrementing the offset value, which is always between 0 and $2^n-1$, will carry from the low order offset field and will decrement the wrap field.

Referring now to Fig. 5, the procedure 70 used to remove data elements from the event queue 30, and the concomitant updates of the head pointer value with each removal, is generally illustrated.

First, a process or thread running on the processor(s) 12 (Fig. 1) will determine if any data elements are on the event queue 30 by checking if the head pointer value 50 (combined with the QBase value 56) points to a non-zero entry. Accordingly, the value at the queue entry 32 pointed to by the head pointer information is pulled from the queue at step 72 and examined at step 74. If the retrieved data element is a zero entry, then the event queue 30 is empty, then the procedure 70 will exit via step 86.
There may be or concurrent threads running, each checking the event queue 30 and attempting to simultaneously remove items from the queue. Therefore, removal of a data element from the event queue preferably is done with an atomic operation that performs a test and conditional store operation to change the head pointer information 50 to point to the next entry in line, such as the compare and store operation performed by the InterlockedCompareExchange function available on the Windows NT operating system and supported by newer microprocessors (e.g., the Pentium Pro manufactured by Intel Corporation). (Windows, and Windows NT are trademarks of Microsoft Corporation of Redmond, Washington; Pentium and Pentium Pro are trademarks of Intel Corporation.) Thus, if the data element pulled from the event queue 30 is non-zero, the procedure 70 moves from the step 74 to the step 76 where a test is made (steps 76, 80) of the entry address (i.e., the head pointer information) to see if it has changed from that used to retrieve the data element from the event queue. If so, such change indicates that the data element has been pulled from the event queue by another process or thread (also operating under the procedure 70), and the procedure for the current executing process or thread will return to step 72 to retrieve and attempt to remove the next data element in the event queue.

If, however the test shows that the entry address remains unchanged, the procedure moves to step 82 to decrement the head pointer information 50 as indicated above. Step 82 is followed by steps 84 and 86 where the procedure 70 is exited and the data value removed used. Step 82 is preferably part of an atomic operation with steps 76 and 80.

The following pseudo code illustrates removal of an data element from the event queue 30. The 64-bit wrapcount/offset is masked and used as an offset into the event queue. As indicated above, to pop a queue entry, the head of the queue is checked for a non-zero entry. If non-zero, the procedure attempts to write the new decremented head pointer with a compare & swap instruction. If the queue is empty, it returns a zero entry in Qitem.
ULONG Qpop(LONGLONG *Qhead, PULONG Qbase, ULONG Qsize)
{
    ULONG Qitem;
    LONGLONG current;
    ULONG mask = Oxffff >> (15-Qsize);
    current=*Qhead;
    Qitem=Qbase[current & mask];
    while(Qitem != 0)
    {
        if (current == *interlockedCompareExchange64(CQHead, --current, current)) break;
        current=*CQHead;
        Qitem=Qbase[current & mask]
    }
    return Qitem;
}

To understand the need for the wrapcount value 52, consider a thread that attempts to remove a data element from the event queue. It first reads the head pointer information and the data element at the head of the event queue, after which it does an atomic compare and swap operation with the head pointer information. But, if in a multitasking environment, the thread presently in the process of removing a data element might get preempted for an arbitrary amount of time between reading the head pointer information 50 and executing the compare and swap. Assume that before the thread is able to complete, the event queue has wrapped completely around and now has the same head pointer offset. By including the wrapcount in the compare and swap operation this problem is minimized. To have the same problem with the wrapcount/head offset included in the same word, the thread would have to be preempted for the exact amount of time for both the offset and wrapcount to rollover to the same value. Since the wrapcount/offset information (i.e., the head pointer information 50) is 64 bits, and it is assumed that a data element will not be removed from the event queue 30 any faster than once per microsecond, a thread would have to be preempted for $2^{64} \times 10^{-6}$ seconds or 585,000 years to hit this race condition. (In contrast, a 32-bit wrapcount/offset value would allow this race condition window to happen in 71 minutes).

The interface unit 20 maintains the tail pointer information that includes the base address (QBase 62) of the queue 30 in memory, the size of the queue (QSize 68'), and the offset (64') into the queue (see Fig. 2B). Since only the interface unit 20 adds data elements to the
queue there is no need for the interface unit 20 to maintain a wrapcount.

When the interface unit 20 adds data element to the tail of the queue 30, it first zeros the entry one beyond the tail. That is, the interface unit always writes two four-byte quantities each time it adds an data element to the queue. First the interface unit 20 writes a zero entry one past the tail, then it writes the new data element to the location pointed to by the tail. The interface unit writes the zero to ensure there is always a zero entry at the tail of the queue. Processes and/or threads running on the processor(s) 12 cannot be relied upon to write this zero entry. Since the process or thread removing data elements from the queue 30 most likely will be running in a user mode, it might be preempted at any time. If a thread is preempted after it has advanced the head pointer, but before it can write a zero to the queue tail, another thread would not know where the queue tail is. The user application or process threads could use a software lock to control their access to the queue. That way, if one thread was preempted at an inopportune time, the other threads wouldn't be able to access the queue. While this would work correctly, it would be inefficient -- a user thread holding the lock can be preempted which stops all threads from using the event queue. The scheme provided by the present invention allows multiple concurrent user threads to access the event queue 30 without using software lock mechanisms and the subsequent priority inversion problem.

Writing the zero entry before writing the new queue data element is the reason the queue grows into lower memory addresses. According to the standard PCI bus protocol, multiple writes are available, starting from the lower address. For the interface unit to ensure the zero entry is written before the new item, it must write the zero to a lower address, hence the queue grows into lower addresses.

An event queue could be made to grow into increasing memory addresses. The interface unit 20 need only write three 4-byte words for each data element added: first the new queue tail would be written; then two entries containing zeros. This ensures there is no window in time in which a zero is not located at the end (tail) of the
event queue 30. However, this approach wastes two queue entries (instead of one) and will require more bus cycles according to conventional PCI protocol.

A special case exists when tail offset value is zero and the queue is about to wrap. The addition of a data element is done in two parts (order is important):

1) First, the 4-byte zero (0000 0000h) data element is written to the address of the entry that forms the top of the event queue 30 (i.e., the highest addressed entry for a queue that grows into lower addresses - before wrapping back to the highest addressed entry);

2) Then, the 4-byte data element is written to the location of the queue entry holding the prior written zero entry - the bottom (i.e., the lowest addressed entry for an event queue that grows into lower addresses) of the queue.

After the data element is added to the event queue, the tail offset is decremented by one (modulo QSize). There is no full or overflow detection; overflows merely wrap the point back to the top.

A novel aspect of the invention is the ability to dynamically resize the event queues during operation. That is, a new event queue can be created, and data element in the event queue being replace are migrated from the old queue to the new while data elements are being added to and removed from the old queue. The basic steps for resizing an event queue are:

1. Allocate the physically contiguous memory for the new queue, and initialize the new queue by zeroing all the entries.

2. Temporarily let users (e.g., threads) of the queue think the queue is empty by setting the head pointer information 50 to point to a zero entry. (This will prevent access because the users will see the non-zero data element.)

3. Update the queue base address in memory, the queue size and the head pointer.

4. Update the tail pointer information maintained by the interface unit 20 so that it points to the new queue.
5. Move any remaining data elements from the old queue to the new queue.

6. If data elements are on the new queue, unblock any users waiting on the queue and let them access the new head pointer.

Figs. 6A through 6G illustrate these steps, and the following discussion explains them in more detail. Fig. 6A illustrates the first step: allocation of memory to the new event queue 90. In this example, the old event queue (92) is 4 entries in size (QSize = 1), and the new queue 90 is eight entries in size (QSize = 2).

The next re-sizing step, illustrated by Fig. 6B, is to temporarily let the user processes/threads think the old event queue 92 is empty by making the head pointer information (e.g., QBase 94 and head offset 96) point to a zero entry - as illustrated in Fig. 6B. The interface unit 20 may still be allowed to add data elements to the old event queue's tail.

As explained above, and as illustrated in Fig. 2A, the head pointer information 50 is made up of the wrapcount/offset which, together with QBase and QSize variables, are used to form the physical address of the queue head entry. Since these values cannot all be updated atomically, they have to be updated in a series of steps so that any user accessing the old event queue during the re-size operation will compute an address that points to a zero entry.

First, a zero is written to the one entry in the old queue that can always be guaranteed to be zero: the item just above the current queue head (i.e. the item with a higher address than the queue head). Since the queue will not overflow, the interface unit 20 will never write anything but a zero to the queue entry just above the head. Then, the head offset value 54 is changed (written) to point to this newly zeroed entry.

When re-pointing the head offset, an atomic compare and swap is preferably used to ensure both that the correct entry was cleared and the old head entry is preserved for a later step in the resize.
operation. A failure of the compare and swap operation will result in simply starting over again with the new head offset.

Next the values for the base address (QBase 94) and queue size (QSize 100) values are modified to reflect those of the new queue, followed by modification of the wrapcount/offset values that make up the head pointer information 50 (Fig. 2A). This is illustrated in Fig. 6C in which the QBase, QSize and head offset values are denoted with reference numerals 94', 96', and 100', and point to the new queue 90. The order of these operations is important because since all three updates are not atomic users of the old queue can be accessing the head pointer during the update and might compute an address with the old offset but the new QBase address value. At each step of the update, the head pointer information must be such to point to an item in the new queue that is zero. As Fig. 6C shows, this step results in the head pointer information 50 pointing to the base of the new queue which contains a zero data element.

The old QBase and QSize 94, 96 values are updated atomically with a 64-bit store. The QBase value is set to the lowest address of the new event queue and QSize value is set to $\log_2(\#\text{entries in new queue})-1$. Since the new queue in this example has 8 entries, QSize is set to 2.

The head pointer information 50 (wrapcount/offset values 52, 54 - Fig. 2A) is also preferably updated atomically with a 64 bit store operation. Before the update, the upper 32 bits of the 64 bit wrapcount/head offset is decremented. This ensures that the new wrapcount/offset will be different from the previous one. Decrementing just the wrapcount portion isn't straightforward since, when the queue is re-sized, the wrapcount portion of the word is shifted. Decrementing the high order 32 bits accomplishes the same goal and is simpler to do.

The head offset value 54 must be set to one more than what the tail offset value 64 (Fig. 2B) will be initialized to. This ensures that new items added by the interface unit 20 to the new queue won't interfere with the old items as they're copied from the old queue onto
the new queue. In this example, the tail offset value 64 will be pointed to entry located at memory address 1C, so the head offset will be set to point to address 00 in this example. The 64-bit head pointer information value 50 should also be changed with an atomic 8 byte write operation.

Next, the tail pointer register 40 is written with a new QBase and offset values 62, 64, to point to the entry location 1C of the new event queue 30 as illustrated in Fig. 6D. The interface unit 20 will not add data elements to the location pointed to by the head pointer information 50 since the user's software will ensures that the queue will not overflow.

If the event queue 30 is being re-sized to a smaller queue (i.e. new queue is smaller than old queue) it is expected that the user's software will ensure that there can be no more than QSize(new queue) data elements on the old event queue before the re-size operation is started.

Now, the re-size operation is ready to proceed to migrating any data elements from the old event queue to the new one. This will require modification of the head offset value 96'. Accordingly, the re-size operation must first determine the number of data elements (QE4, QE5, QE6) remaining in the old event queue 92. This can only be done now since the interface unit 20, at this point of the process, had been adding data elements to the old queue. The number of data elements on the old queue is just the number of entries containing those elements between (that is, counting down from) the data element pointed to by the old values for QBase and offset (94, 96) head pointer information value 50 (in this example, "Old Head" 100) and the first zero entry. Call this NumOldEntries, which in this example is 3.

Update the head offset value 96' so that it points to NumOldEntries more than the original value of the tail offset. In this example the new queue's original tail offset points to entry 1C and NumOldEntries is 3. The Head offset is then set to point to entry 08. Then, any remaining data elements are moved from the old queue to
the new queue. The move operation is performed so that the oldest data elements from the old event queue (QE4 in this example) is moved last. After this last move, the head offset value 96' will be pointing to the oldest item from the old event queue and the new queue is fully functional. Users (processes/threads) will now see a non-zero item at the head of the queue and may start pulling data elements from the queue.

The interface unit 20 will simultaneously add data elements to the new queue. But since the queue won't overflow, additions to the new event queue won't conflict with the copying of the data elements from the old queue.

During the event queue re-size operation the head pointer information 50 (i.e., the off set value 54) was set to a value that pointed to a zero data element so that users accessing the queue for removal of data elements would think the queue was empty. As a result, in a multitasking environment, there may be threads blocked even though the event queue is actually not empty. As the last step in the re-size operation, the appropriate number of threads should be unblocked if there are data elements on the queue.

Finally, the memory used by the old queue 92 (Fig. 6F) must be deallocated (Fig. 6G). However, If the old queue 92 is simply deallocated there is a potential problem. Consider a thread in the process of examining the old queue for a new entry. It computes the head entry address as discussed above, and is then preempted. When the thread restarts the new queue has been created and the old queue deallocated. The thread will get a memory protection error when it reads what it thinks is the address for the head of the queue. The thread would never use the data from this read -the compare and swap would fail and the thread would simply compute a new head pointer (which would point to the new queue) and try again.

There may be cases in which the event queue could overflow, i.e., the addition of data elements begins to overwrite the older data elements stored in the event queue 30 and awaiting removal. Detection and prevention of overflow can be accomplished by having the kernel
agent (i.e., operating system of the processor(s) 12)) periodically writing the head pointer to a memory register (not shown) of the interface unit 20. Interface unit will then be able to periodically check the memory register containing the head pointer, and compare it with the content of the tail pointer register 40 in order to not let the tail of the event queue 30 not pass the most recently written value of the head pointer.
WHAT IS CLAIMED IS:

1. A method of managing a data structure in a memory having a number of storage location for storing data elements received from a first processing unit, the data elements being retrieved by a second processing unit, the method including the steps of:
   providing a pointer value indicative of a location of a data element next to be retrieved from the data structure;
   retrieving the data element next to be retrieved by:
      (a) using the pointer value to retrieve the data element,
      (b) checking the data element and if non-zero, performing an atomic operation to conditionally store to the memory a modified pointer value indicative of a next to be retrieved data element if a test of the address data is unchanged,
      (c) if the test indicates the pointer value is changed, performing steps (a)-(c) again, and retaining the data element if non-zero.
FIG. 2A.

FIG. 2B.

FIG. 3A.

FIG. 3B.

FIG. 3C.
ENTRY ADDRESS FORMATION

HEAD POINTER  63  WRAPCOUNT  QSIZE  OFFSET

0xFFFF >> (15 - QSIZE)  AND

<<2

31  QBASE

+  ADDRESS OF Q ENTRY

FIG. 4.

REMOVING ENTRIES FROM QUEUE

RETRIEVE AND CHECK

ENTRY ZERO?

YES

NO

TEST ENTRY ADDRESS

CHANGE

YES

NO

DECREMENT ENTRY HEAD POINTER

ACT ON ENTRY

EXIT

FIG. 5.

SUBSTITUTE SHEET (RULE 26)
### A. CLASSIFICATION OF SUBJECT MATTER

| IPC  | G06F5/06 |

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)

| IPC  | 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)

### C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>&quot;CONCURRENT ALGORITHM FOR MANAGING A FIRST-IN, FIRST-OUT QUEUE WITH TWO-WAY POINTERS&quot; IBM TECHNICAL DISCLOSURE BULLETIN, vol. 34, no. 6, 1 November 1991, pages 80-88, XP000228358 see page 86, line 12 - line 18</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>&quot;MULTI-ACCESS FIRST-IN-FIRST-OUT QUEUE USING 370 COMPARE AND SNAP&quot; IBM TECHNICAL DISCLOSURE BULLETIN, vol. 36, no. 2, 1 February 1993, pages 327-330, XP000354355 see the whole document</td>
<td>1</td>
</tr>
</tbody>
</table>

Further documents are listed in the continuation of box C.

Patent family members are listed in 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.
  * A: document member of the same patent family

**Date of the actual completion of the international search:** 28 April 1999

**Date of mailing of the international search report:** 07/05/1999

**Name and mailing address of the ISA:**
European Patent Office, P.B. 5818 Patentlaan 2 NL-2280 HV Rijswijk Tel: (+31-70) 340-2040, Tx: 31 651 spc nl, Fax: (+31-70) 340-3016

**Authorized officer:** Cohen, B
<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>&quot;MECHANISM FOR MULTIPLE SOURCE ACCESS OF QUEUES AND STACKS&quot;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IBM TECHNICAL DISCLOSURE BULLETIN, vol. 38, no. 6, 1 June 1995, pages 99-107,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XP000520596 see page 104, paragraph 3</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>EP 0 570 877 A (ALCATEL NV)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>24 November 1993 see column 6, line 16 - line 37</td>
<td></td>
</tr>
<tr>
<td>Patent document cited in search report</td>
<td>Publication date</td>
<td>Patent family member(s)</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>-----------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AT 176112 T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CA 2096717 A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DE 69323111 D</td>
</tr>
</tbody>
</table>