US008407444B2

(12) **United States Patent**
Allen et al.

(10) **Patent No.: US 8,407,444 B2**
(45) **Date of Patent: *Mar. 26, 2013**

(54) **HARDWARE OFF-LOAD GARBAGE COLLECTION ACCELERATION FOR LANGUAGES WITH FINALIZERS**

(75) Inventors: **Joseph H. Allen**, Belmont, MA (US); **Moshe M. E. Matsa**, Cambridge, MA (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 499 days.

This patent is subject to a terminal disclaimer.

(21) Appl. No.: **12/645,537**

(22) Filed: **Dec. 23, 2009**

(65) **Prior Publication Data**

US 2011/0153691 A1 Jun. 23, 2011

(51) **Int. Cl.**
G06F 12/02 (2006.01)

(52) **U.S. Cl.** **711/170**; 711/166; 711/E12.002; 707/813

(58) **Field of Classification Search** 711/170, 711/172, E12.001, E12.002, 166; 707/813, 707/814, 816, 999.206
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

6,336,180 B1 1/2002 Long et al.
6,473,777 B1 10/2002 Hendler et al.
6,604,182 B1 * 8/2003 Sexton et al. 711/170
6,920,541 B2 7/2005 Kolodner et al.
6,950,837 B2 9/2005 Subramoney et al.
7,043,623 B2 5/2006 Chen et al.
7,069,279 B1 6/2006 Rau et al.
7,200,741 B1 4/2007 Mine
7,228,543 B2 * 6/2007 Baylis 718/1
7,302,544 B2 11/2007 Chung et al.
7,412,580 B1 8/2008 Garthwaite
7,822,938 B2 * 10/2010 Dussud et al. 711/170
2003/0196061 A1 * 10/2003 Kawahara et al. 711/170
2008/0195682 A1 * 8/2008 Holt 707/206
2008/0263295 A1 10/2008 Printezis et al.
2008/0320485 A1 12/2008 Joffe et al.
2009/0006506 A1 1/2009 DiFlora
2009/0055603 A1 2/2009 Holt

(Continued)

OTHER PUBLICATIONS

Tamar Domani et al., Implementing an On-the-fly Garbage Collector for Java, Article, ACM SIGPLAN Notices, 2001, pp. 1-15, vol. 36, Issue 1, Association for Computing Machinery (ACM), New York, NY.

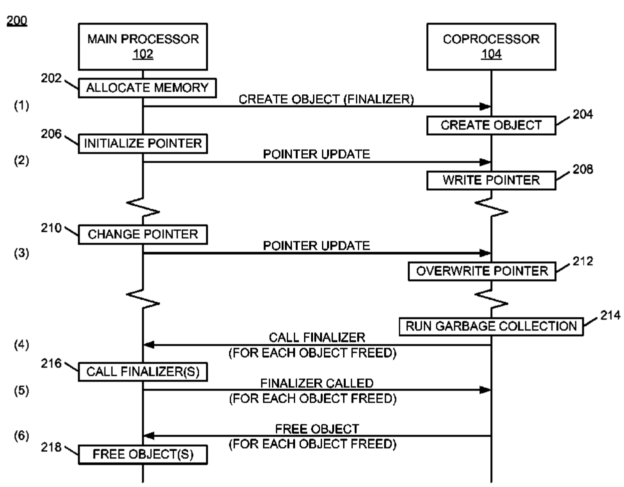
(Continued)

Primary Examiner — Pierre-Michel Bataille
(74) *Attorney, Agent, or Firm* — Lee Law, PLLC; Christopher B. Lee

(57) **ABSTRACT**

A memory allocation message for each primary memory allocation in a primary memory made by a primary processor is received at a hardware memory management module, including an indication of whether a finalizer routine is associated with each primary memory allocation. A representation of each primary memory allocation is allocated within a second memory in response to each memory allocation message, including the indication of whether there is the associated finalizer routine. A determination is made, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the primary memory. A call object finalizer message is sent to the primary processor instructing the primary processor to call the finalizer routine associated with the primary memory allocation in the primary memory in response to determining that the primary memory allocation has the associated finalizer routine.

20 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

2010/0082930 A1* 4/2010 Jiva et al. 711/166

OTHER PUBLICATIONS

Author Unknown, TADS 3 System Manual: Automatic Garbage Collection and Finalization, Webpage/site: 1. http://www.tads.org/t3doc/doc/sysman/gc.htm, Printed from Website on Dec. 9, 2009, pp. 1-4, Published on the World Wide Web.

John N. Zigman, Stephen M. Blackburn, Java Finalize Method, Orthogonal Persistence and Transactions, Article, Proceedings of the 8th International Workshop on Persistent Object Systems (POS8), 1999, pp. 1-7, Morgan Kaufmann Publishers Inc., San Francisco, CA.

Scott Nettles, James O'Toole, Real-Time Replication Garbage Collection, Article, Conference on Programming Language Design and Implementation, 1993, pp. 1-10, Association for Computing Machinery (ACM), New York, NY.

James O'Toole, Scott Nettles, Concurrent Replicating Garbage Collection, Article, Conference on LISP and Functional Programming, 1994, pp. 1-8, Association for Computing Machinery (ACM), New York, NY.

Jeffrey Richter, Garbage Collection: Automatic Memory Management in the Microsoft .NET Framework, Article, MSDN Magazine, Dec. 2000, pp. 1-18, Microsoft Corporation, Published on the World Wide Web.

Matthias Meyer, An On-Chip Garbage Collection Coprocessor for Embedded Real-Time Systems, Article, Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Aug. 2005, pp. 517-524, IEEE Computer Society, Hong Kong.

Kevin D. Nilsen, Cost-Effective Hardware-Assisted Real-Time Garbage Collection, Article, ACM SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 1-10, Association for Computing Machinery (ACM), New York, NY.

Kevin Nilsen, A High-Performance Architecture for Real-Time Garbage Collection, Article, Submitted to OOPSLA Workshop on Garbage Collection in Object-Oriented Systems, 1991, pp. 1-5, Published on the World Wide Web.

United States Patent and Trademark Office, Office Action for U.S. Appl. No. 12/645,514, Feb. 21, 2012, pp. 1-23, Alexandria, VA, USA.

Chen-Yong Cher, et al., Cell GC: Using the Cell Synergistic Processor as a Garbage Collection Coprocessor, Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, Mar. 5, 2008, pp. 1-10, Association for Computing Machinery, New York, NY, USA.

United States Patent and Trademark Office, Office Action for U.S. Appl. No. 12/645,514, May 31, 2012, pp. 1-26, Alexandria, VA, USA.

* cited by examiner

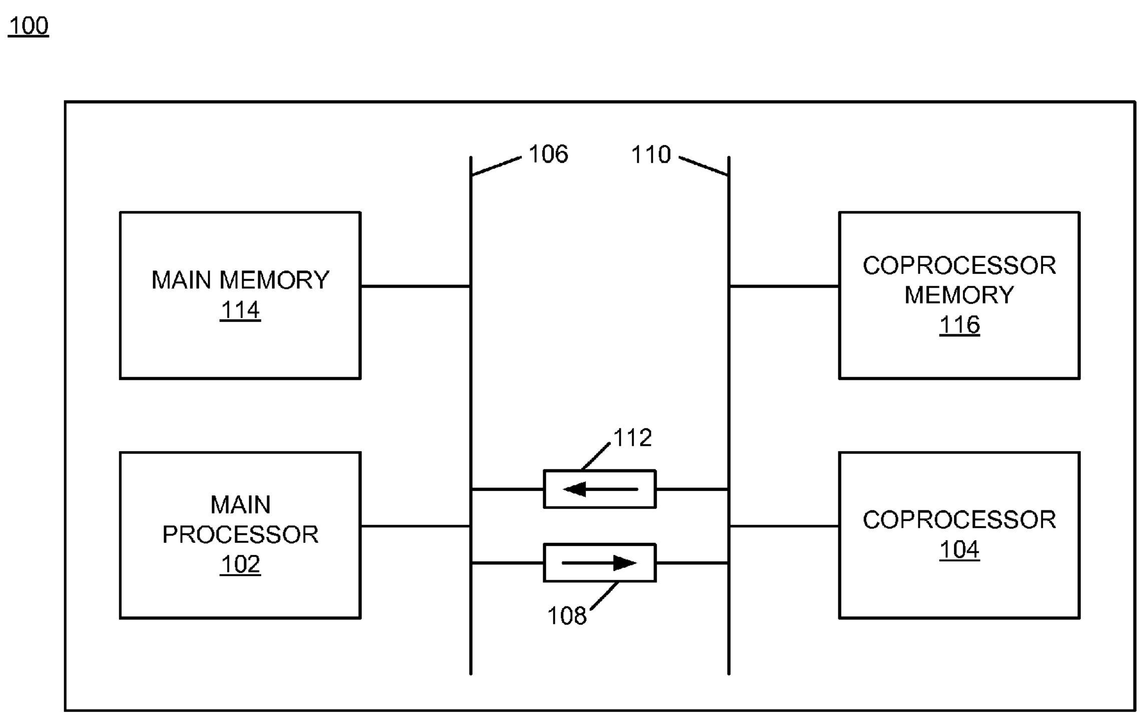

100
106
110
MAIN MEMORY
114
COPROCESSOR
MEMORY
116
112
MAIN
PROCESSOR
102
COPROCESSOR
104
108

FIG.1

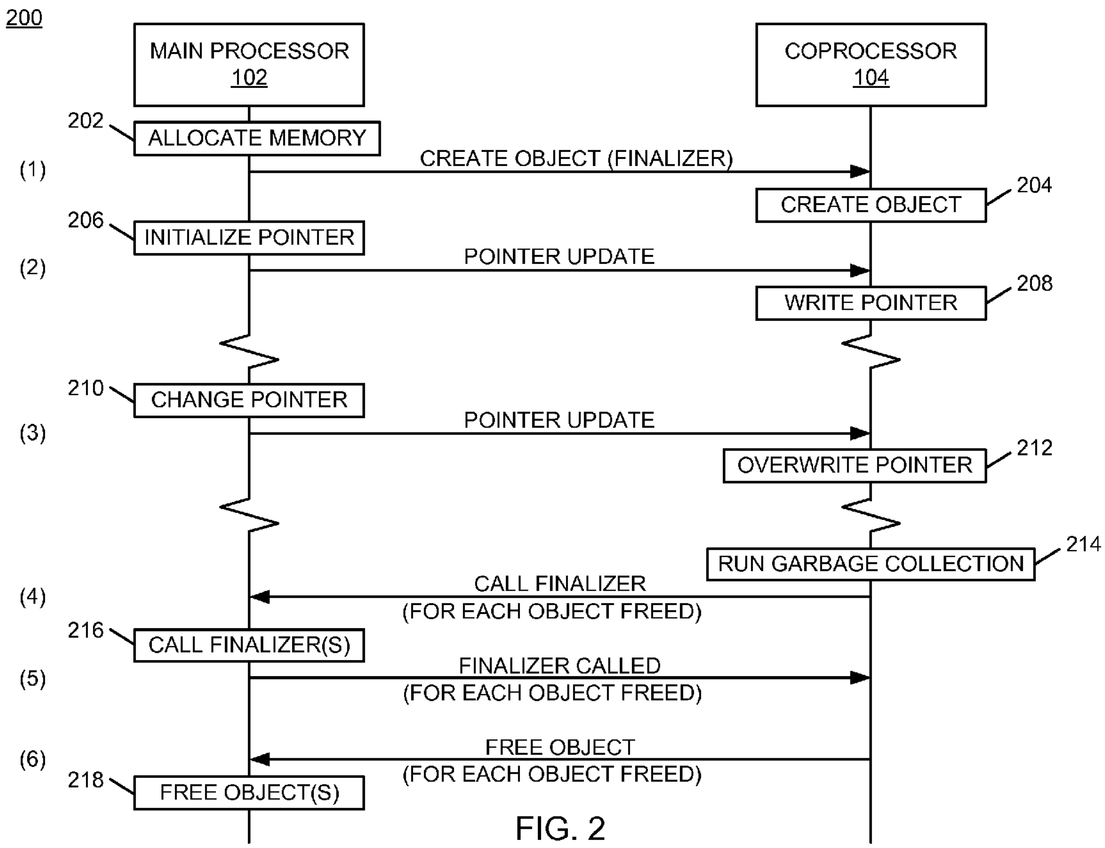

200
MAIN PROCESSOR 102
COPROCESSOR 104
202
ALLOCATE MEMORY
CREATE OBJECT (FINALIZER)
(1)
CREATE OBJECT
204
206
INITIALIZE POINTER
POINTER UPDATE
(2)
WRITE POINTER
208
210
CHANGE POINTER
POINTER UPDATE
(3)
OVERWRITE POINTER
212
RUN GARBAGE COLLECTION
214
CALL FINALIZER
(FOR EACH OBJECT FREED)
(4)
216
CALL FINALIZER(S)
FINALIZER CALLED
(FOR EACH OBJECT FREED)
(5)
FREE OBJECT
(FOR EACH OBJECT FREED)
(6)
218
FREE OBJECT(S)

FIG. 2

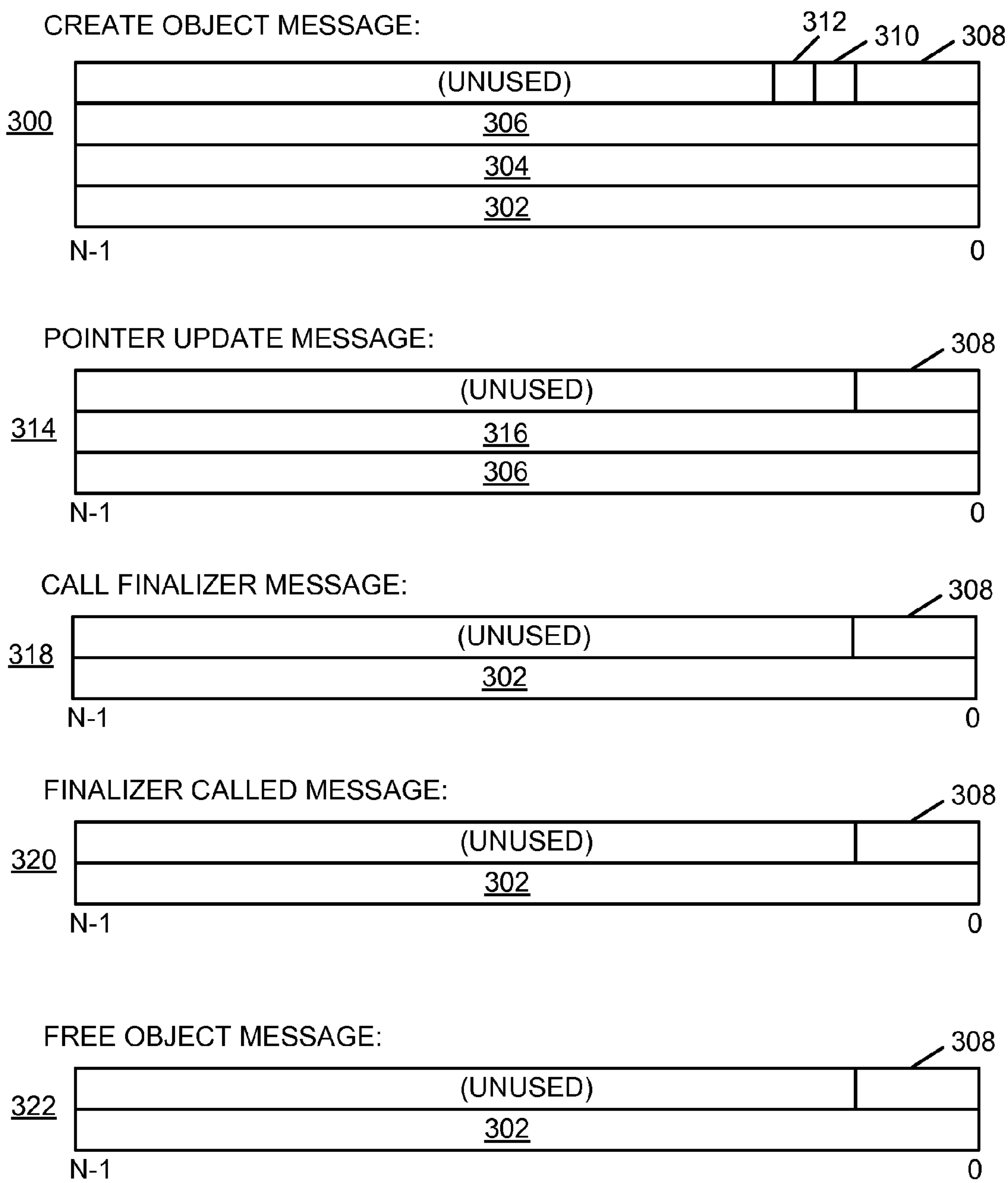

CREATE OBJECT MESSAGE:
312
310
308
(UNUSED)
300
306
304
302
N-1
0
POINTER UPDATE MESSAGE:
308
(UNUSED)
314
316
306
N-1
0
CALL FINALIZER MESSAGE:
308
(UNUSED)
318
302
N-1
0
FINALIZER CALLED MESSAGE:
308
(UNUSED)
320
302
N-1
0
FREE OBJECT MESSAGE:
308
(UNUSED)
322
302
N-1
0

FIG. 3

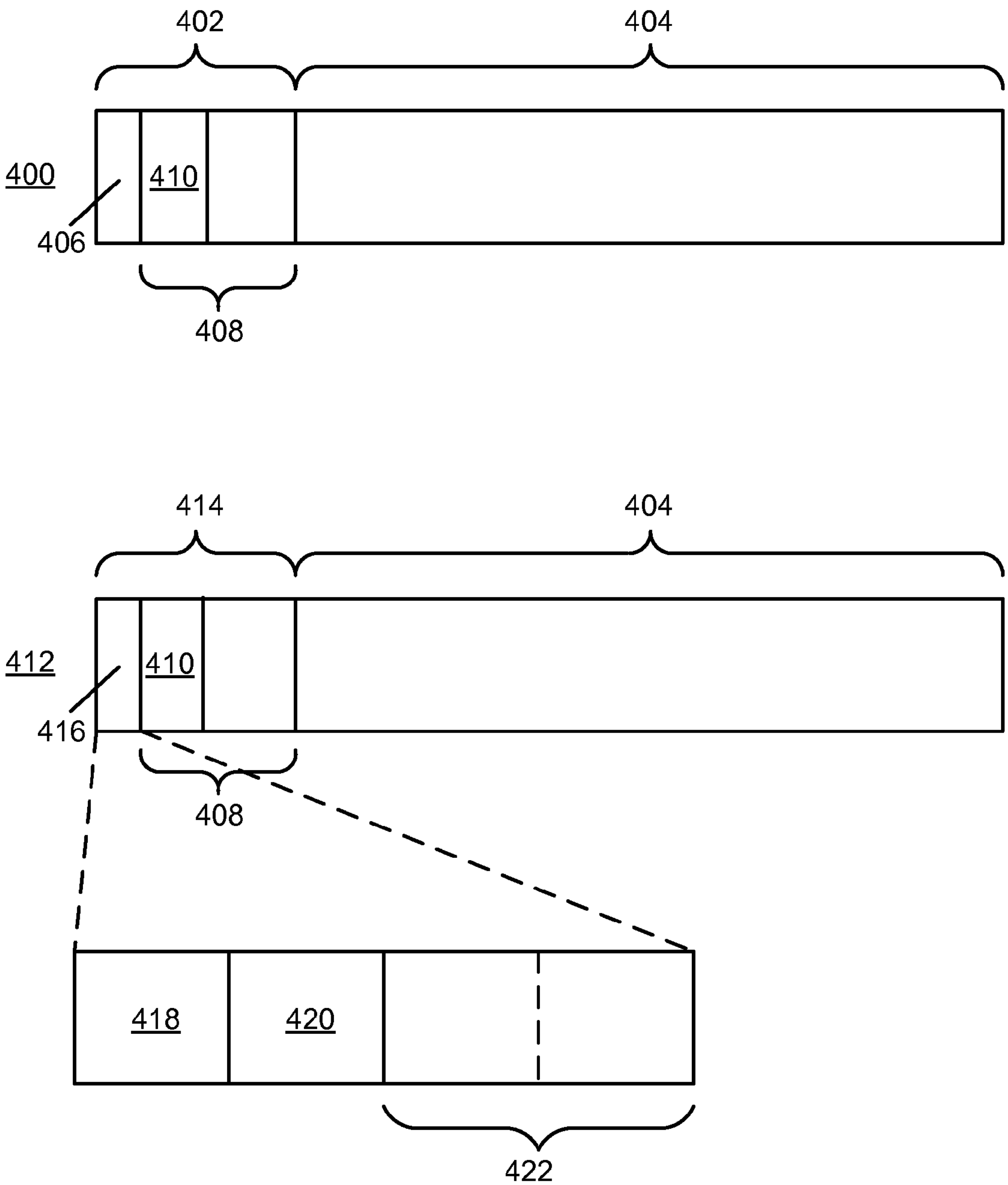


FIG. 4

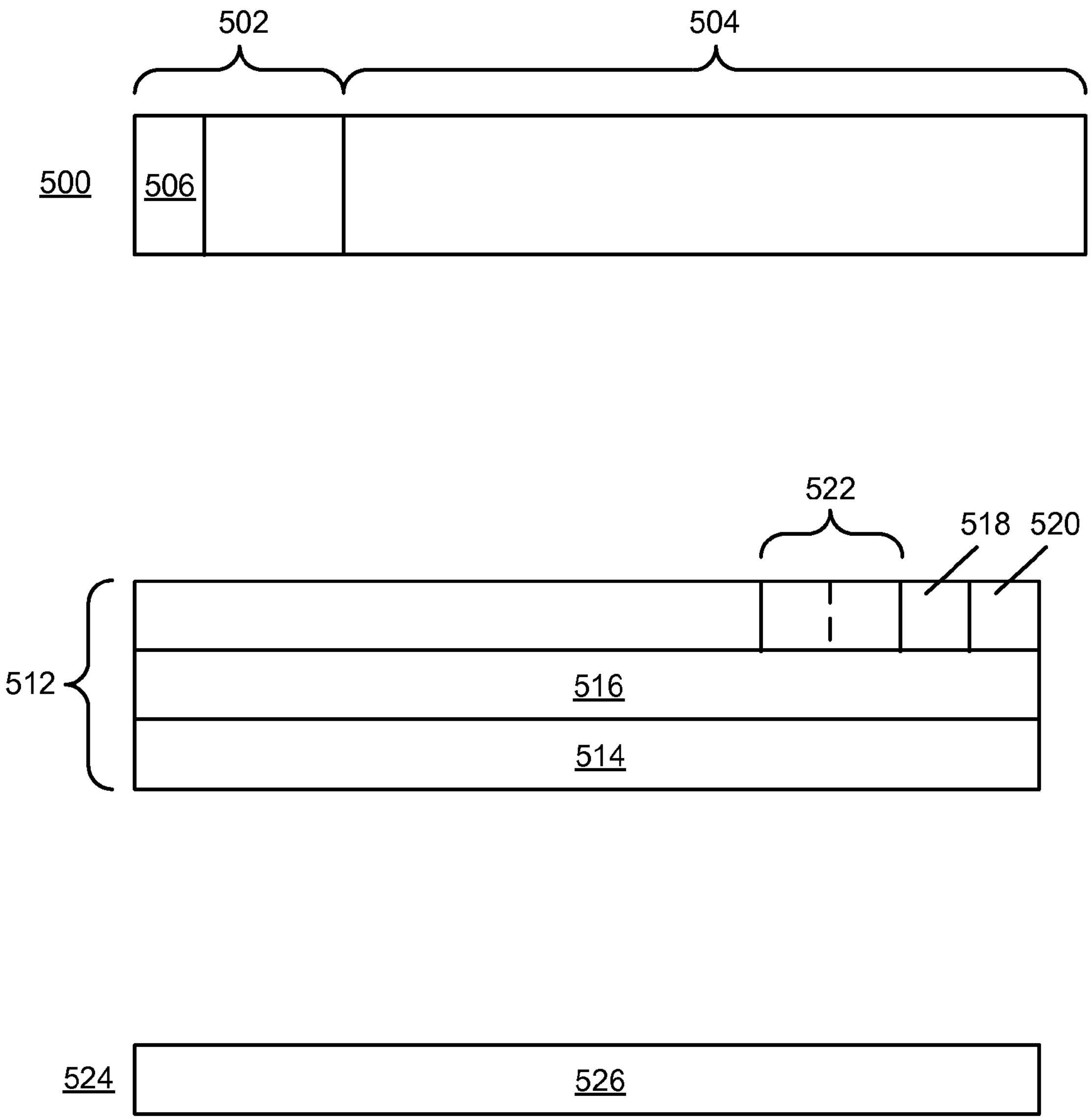


FIG. 5

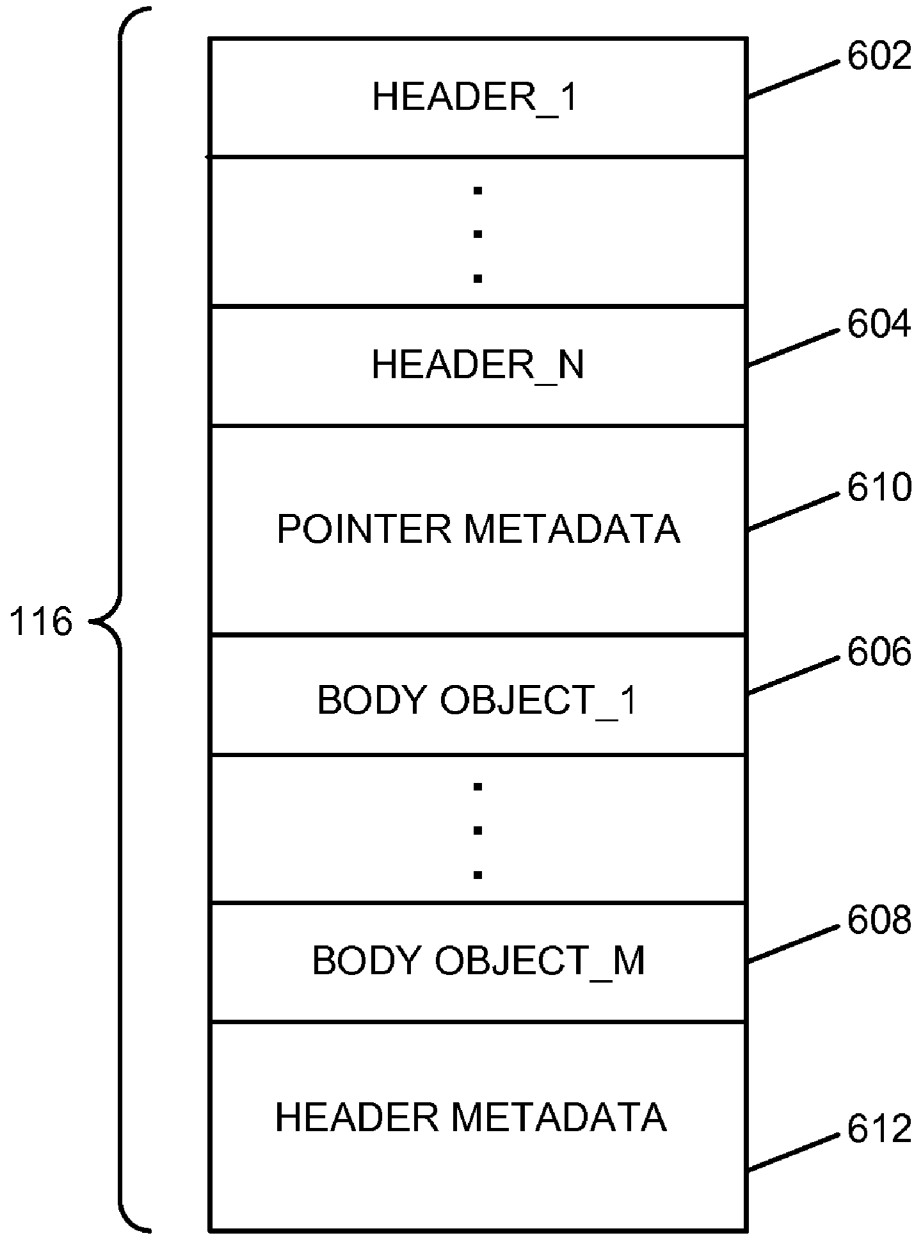

HEADER_1
602
HEADER_N
604
POINTER METADATA
610
116
BODY OBJECT_1
606
BODY OBJECT_M
608
HEADER METADATA
612

FIG. 6

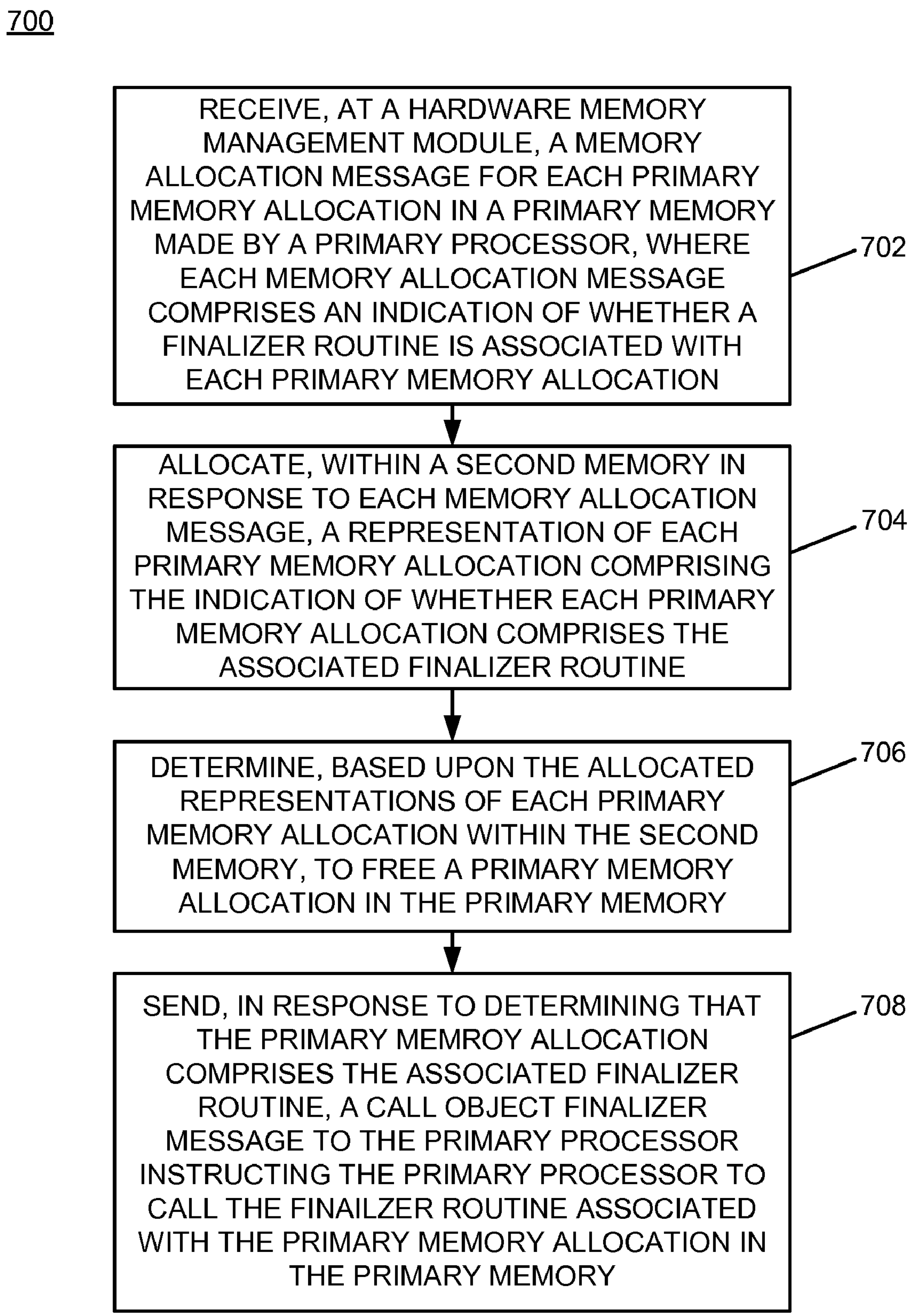
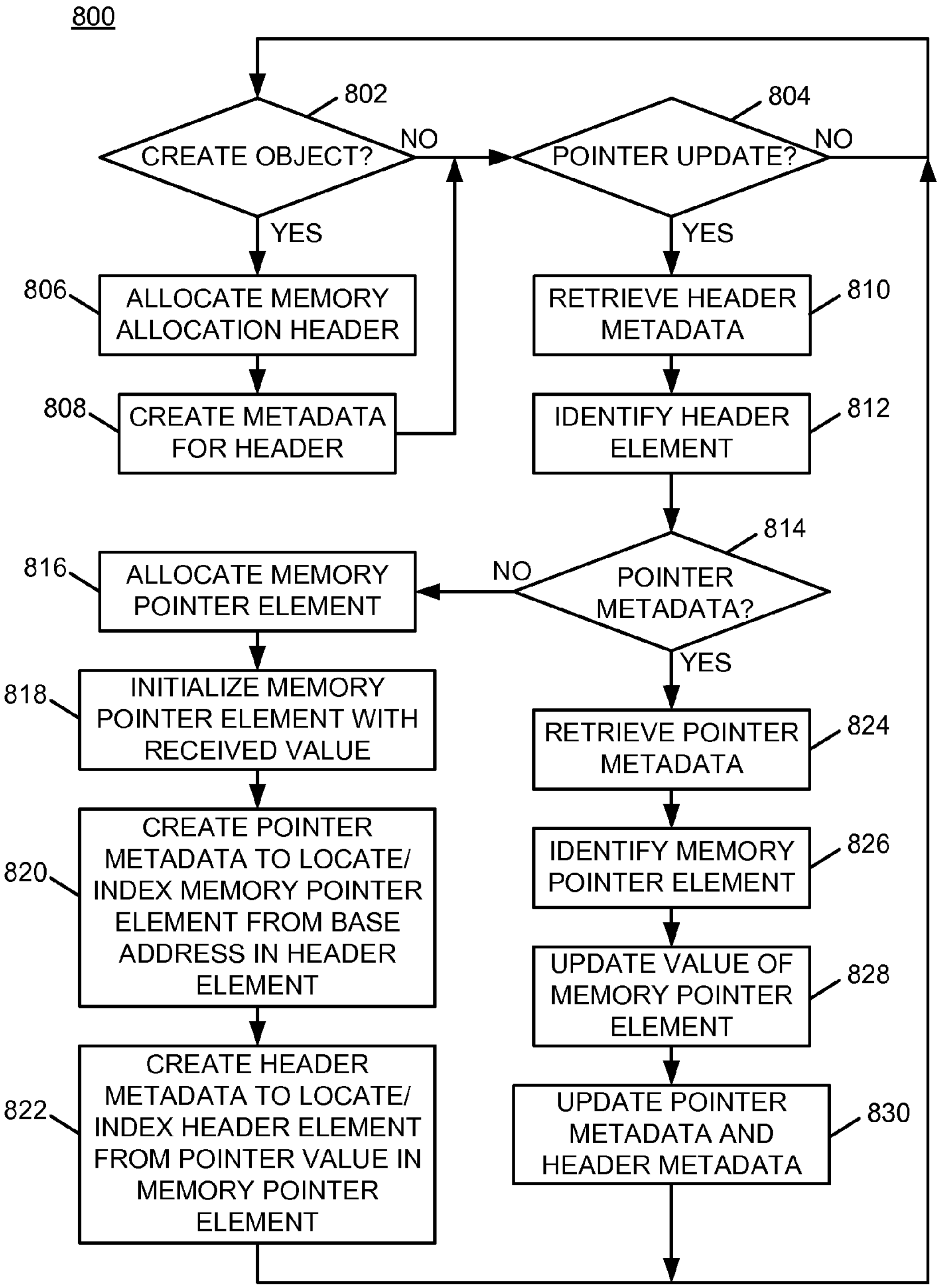

700
RECEIVE, AT A HARDWARE MEMORY MANAGEMENT MODULE, A MEMORY ALLOCATION MESSAGE FOR EACH PRIMARY MEMORY ALLOCATION IN A PRIMARY MEMORY MADE BY A PRIMARY PROCESSOR, WHERE EACH MEMORY ALLOCATION MESSAGE COMPRISES AN INDICATION OF WHETHER A FINALIZER ROUTINE IS ASSOCIATED WITH EACH PRIMARY MEMORY ALLOCATION
702
ALLOCATE, WITHIN A SECOND MEMORY IN RESPONSE TO EACH MEMORY ALLOCATION MESSAGE, A REPRESENTATION OF EACH PRIMARY MEMORY ALLOCATION COMPRISING THE INDICATION OF WHETHER EACH PRIMARY MEMORY ALLOCATION COMPRISES THE ASSOCIATED FINALIZER ROUTINE
704
DETERMINE, BASED UPON THE ALLOCATED REPRESENTATIONS OF EACH PRIMARY MEMORY ALLOCATION WITHIN THE SECOND MEMORY, TO FREE A PRIMARY MEMORY ALLOCATION IN THE PRIMARY MEMORY
706
SEND, IN RESPONSE TO DETERMINING THAT THE PRIMARY MEMROY ALLOCATION COMPRISES THE ASSOCIATED FINALIZER ROUTINE, A CALL OBJECT FINALIZER MESSAGE TO THE PRIMARY PROCESSOR INSTRUCTING THE PRIMARY PROCESSOR TO CALL THE FINAILZER ROUTINE ASSOCIATED WITH THE PRIMARY MEMORY ALLOCATION IN THE PRIMARY MEMORY
708

FIG. 7



800
CREATE OBJECT?
802
NO
YES
POINTER UPDATE?
804
NO
YES
806
ALLOCATE MEMORY ALLOCATION HEADER
808
CREATE METADATA FOR HEADER
RETRIEVE HEADER METADATA
810
IDENTIFY HEADER ELEMENT
812
POINTER METADATA?
814
NO
YES
816
ALLOCATE MEMORY POINTER ELEMENT
818
INITIALIZE MEMORY POINTER ELEMENT WITH RECEIVED VALUE
820
CREATE POINTER METADATA TO LOCATE/ INDEX MEMORY POINTER ELEMENT FROM BASE ADDRESS IN HEADER ELEMENT
822
CREATE HEADER METADATA TO LOCATE/ INDEX HEADER ELEMENT FROM POINTER VALUE IN MEMORY POINTER ELEMENT
RETRIEVE POINTER METADATA
824
IDENTIFY MEMORY POINTER ELEMENT
826
UPDATE VALUE OF MEMORY POINTER ELEMENT
828
UPDATE POINTER METADATA AND HEADER METADATA
830

FIG. 8

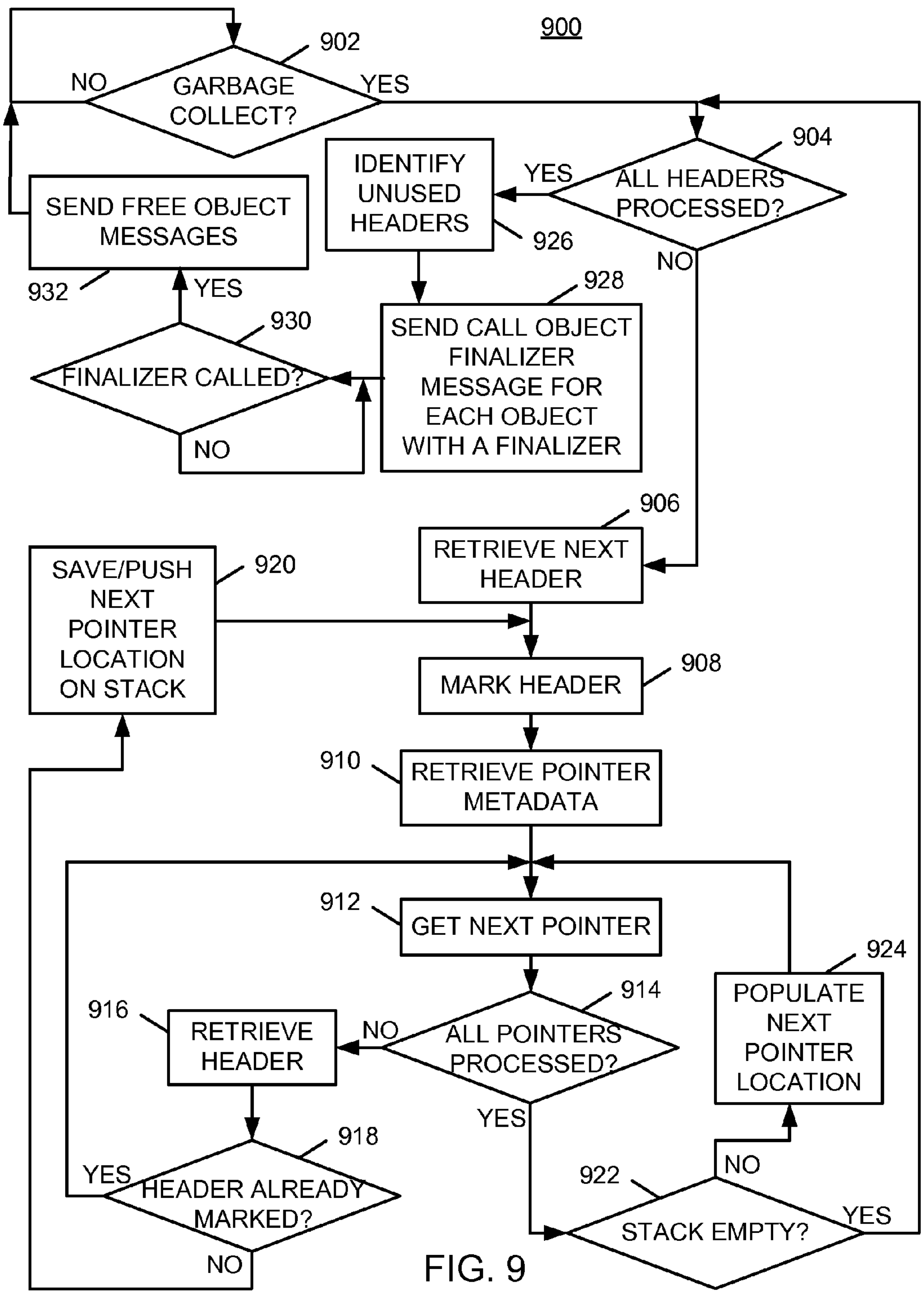

900
902
GARBAGE COLLECT?
NO
YES
904
ALL HEADERS PROCESSED?
YES
NO
IDENTIFY UNUSED HEADERS
926
SEND FREE OBJECT MESSAGES
932
YES
928
SEND CALL OBJECT FINALIZER MESSAGE FOR EACH OBJECT WITH A FINALIZER
930
FINALIZER CALLED?
NO
906
RETRIEVE NEXT HEADER
SAVE/PUSH NEXT POINTER LOCATION ON STACK
920
MARK HEADER
908
910
RETRIEVE POINTER METADATA
912
GET NEXT POINTER
924
POPULATE NEXT POINTER LOCATION
916
RETRIEVE HEADER
NO
914
ALL POINTERS PROCESSED?
YES
918
YES
HEADER ALREADY MARKED?
NO
922
NO
STACK EMPTY?
YES

FIG. 9

HARDWARE OFF-LOAD GARBAGE COLLECTION ACCELERATION FOR LANGUAGES WITH FINALIZERS

RELATED APPLICATIONS

This application is related to concurrently filed U.S. utility patent application Ser. No. 12/645,514, titled "HARDWARE OFF-LOAD MEMORY GARBAGE COLLECTION ACCELERATION," which has a current status of pending, and which is incorporated herein by reference in its entirety.

BACKGROUND

The present invention relates to memory garbage collection. More particularly, the present invention relates to hardware off-load garbage collection acceleration for languages that include finalizers, such as Java® or other garbage collected languages that include finalizers.

Conventional memory garbage collection is performed by a processor to determine memory allocations that are no longer needed by the processor. A processor executes an algorithm, known as a garbage collection algorithm, to identify the memory allocations that it no longer needs. Examples of garbage collection algorithms include a mark and sweep garbage collection algorithm and a reference counting garbage collection algorithm.

BRIEF SUMMARY

A method includes receiving, at a hardware memory management module, a memory allocation message for each primary memory allocation in a primary memory made by a primary processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation; allocating, within a second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine; determining, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the primary memory; and sending, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the primary processor instructing the primary processor to call the finalizer routine associated with the primary memory allocation in the primary memory.

A system includes a first processor operatively coupled to a first memory; a bi-directional message queue; and an off-load processor operatively coupled to a second memory, and programmed to: receive, via the bi-directional message queue, a memory allocation message for each primary memory allocation in the first memory made by the first processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation; allocate, within the second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine; determine, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the first memory; and send, via the bi-directional message queue, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the first processor instructing the first processor to call the finalizer routine associated with the primary memory allocation in the first memory.

A computer program product includes a computer useable storage medium including computer readable program code, wherein the computer readable program code when executed on a computer causes the computer to: receive a memory allocation message for each primary memory allocation in a first memory made by a first processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation; allocate, within a second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine; determine, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the first memory; and send, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the first processor instructing the first processor to call the finalizer routine associated with the primary memory allocation in the first memory.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

FIG. **1** is a block diagram of an example of an implementation of a system for automated hardware off-load garbage collection acceleration for languages that include finalizers according to an embodiment of the present subject matter;

FIG. **2** is a message flow diagram of an example of an implementation of a messaging interaction between a main processor and a coprocessor for automated hardware off-load garbage collection acceleration for languages that include finalizers according to an embodiment of the present subject matter;

FIG. **3** is an illustration of example implementations of message formats that may be used for messages described in association with FIG. **2** according to an embodiment of the present subject matter;

FIG. **4** is a block diagram of an example of an implementation of memory allocations for direct mapping between a main memory and a coprocessor memory according to an embodiment of the present subject matter;

FIG. **5** is a block diagram of an example of an implementation of memory allocations for reduced mapping within the coprocessor memory relative to memory allocations within the main memory according to an embodiment of the present subject matter;

FIG. **6** is a block diagram of an example of an implementation of memory allocations and associated metadata for reduced mapping within the coprocessor memory according to an embodiment of the present subject matter;

FIG. **7** is a flow chart of an example of an implementation of a process for automated hardware off-load garbage collection acceleration for languages that include finalizers according to an embodiment of the present subject matter;

FIG. **8** is a flow chart of an example of an implementation of a process for automated hardware off-load garbage collection acceleration for languages that include finalizers to receive and process messages for memory element creation and pointer updating according to an embodiment of the present subject matter; and

FIG. **9** is a flow chart of an example of an implementation of a process for automated hardware off-load garbage collection acceleration for languages that include finalizers using a processor memory stack to instruct a main processor to free memory allocations according to an embodiment of the present subject matter.

DETAILED DESCRIPTION

The examples set forth below represent the necessary information to enable those skilled in the art to practice the invention and illustrate the best mode of practicing the invention. Upon reading the following description in light of the accompanying drawing figures, those skilled in the art will understand the concepts of the invention and will recognize applications of these concepts not particularly addressed herein. It should be understood that these concepts and applications fall within the scope of the disclosure and the accompanying claims.

The subject matter described herein provides hardware off-load garbage collection acceleration for languages that include finalizers, such as Java® or other garbage collected languages that include finalizers. It should be noted that Java® is a registered trademark of Sun Microsystems, Inc. A system for hardware off-load garbage collection acceleration for languages that include finalizers includes a main (e.g., primary) processor and a coprocessor that operates as a hardware off-load module. Performance improvements for the main processor involve off-loading garbage collection activities to the coprocessor. Rather than performing its own memory management for garbage collection, the main processor communicates with the coprocessor to inform the coprocessor of memory allocation and memory update activities. The main processor also communicates finalizer association information, such as Java® finalizer association information, to the coprocessor to identify each memory allocation that has an associated finalizer routine. The coprocessor independently and autonomously executes garbage collection activities and identifies memory allocations that are no longer used by the main processor. The coprocessor communicates with the main processor to instruct the main processor to either call finalizer routines, such as Java® finalizer routines, for memory allocations that have associated finalizer routines or to free the identified memory allocations that are no longer in use. If the memory allocation has an associated finalizer routine, the coprocessor instructs the main processor to free the identified memory allocations that are no longer in use in response to receipt of a finalizer called message from the primary processor. As such, the time consuming task of memory garbage collection is partitioned from the main processor and performance for the main processor may be improved. It should be noted that in a different embodiment, the main processor and co-processor may be separate threads and the separate threads may be running on separate processors of a multi-processor system.

The main processor is coupled to a main (e.g., primary) memory and the coprocessor is coupled to a second memory. The main processor executes one or more applications and allocates memory for the various applications within the main memory. Rather than performing its own memory management for garbage collection, the main processor additionally sends a memory allocation message for each primary memory allocation in the main memory to the coprocessor. Each memory allocation message includes an indication of whether a finalizer routine is associated with each primary memory allocation. The coprocessor receives the memory allocation message and allocates a representation of each primary memory allocation within the second memory in response to each received memory allocation message including the indication of whether each primary memory allocation has an associated finalizer routine. The coprocessor executes a garbage collection algorithm concurrently with the main processor actively processing application actions. The coprocessor identifies memory allocations within the second memory that are no longer in use by the main processor and sends, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the primary processor instructing the primary processor to call the finalizer routine associated with the primary memory allocation in the primary memory. Alternatively, if the memory allocation has no associated finalizer routine, or in response to a message from the main processing indicating that the finalizer routine has been called, a memory free message is sent to the primary processor instructing the primary processor to free the main memory allocation in the main memory.

The main processor and the coprocessor are separated and communicate via a bi-directional message queue. The bi-directional message queue may be implemented using two uni-directional message buffers/queues. Each processor has write access to one of the two uni-directional message queues and read access to the other message queue. The respective uni-directional message queues may be memory mapped, register mapped, or otherwise accessible for write and read operations, respectively, by the respective processors. When either processor wishes to communicate with the other, it writes a message to its respective write message queue. On the opposite receive side for each message queue, the respective other processor reads messages written to its respective read message queue. The uni-directional message queues may be organized as a first in first out (FIFO) queue structure. As such, the first message written to one of the message queue by one of the processors will be the first message read by the other processor. It should be noted that in a different embodiment, the bi-directional message queues may be implemented in shared memory and software or may use some other dedicated message passing hardware which may be available in a multi-processor system.

In one example implementation, the main memory and the second memory may be implemented with memory devices that are the same size. This form of implementation shall be referred to herein as “direct mapping” between the main memory and the second memory. In this example, the coprocessor may allocate a memory object identical to the memory object allocated by the main processor. Memory allocations within the coprocessor may include metadata (e.g., memory management bits, flags, or other metadata) to allow the coprocessor to identify the unused memory allocations in the main memory. The metadata may include identifiers to represent items as root set memory objects (e.g., static or global variables or pointers, stack variables or pointers, or other root set memory objects). Memory allocations within the main memory may include reserved storage for the metadata used by the coprocessor to provide the storage location(s) for the metadata created and used by the coprocessor.

In an alternative example implementation, the coprocessor may implement a set of data structures for managing memory allocations in the main memory instead of allocating memory objects that are identical to those allocated by the main processor. In such an implementation, a second memory smaller than the main memory may be used. The set of data structures may include a memory allocation header element and a memory pointer element. This form of implementation shall be referred to herein as "reduced mapping" within the second memory relative to the main memory.

The memory allocation header element may be allocated within the second memory in response to a memory allocation message (e.g., a create object message) received from the main processor. The memory allocation message may include a base address, a size of the memory allocation within the main memory, an indication of whether the object is a root set memory object, and an indication of whether a finalizer routine is associated with the memory allocation. The memory allocation header element may be created to include the base address, the size of the main memory allocation within the main memory, the indication of whether the object is a root set memory object, and the indication of whether a finalizer routine is associated with the memory allocation. Additional storage for garbage collection (e.g., a mark bit for use during garbage collection) may be added to the created memory allocation header element for each memory allocation within the main memory.

When the main processor initializes a memory allocation as a pointer, it may send a pointer initialization message (e.g., a pointer update message) to the coprocessor. The pointer initialization message may include an address within the main memory (e.g., the pointer "value") to which the pointer has been initialized. The coprocessor may create a memory pointer element within the second memory to represent the initialized pointer in the main memory. The coprocessor may create one or more (e.g., multiple) memory pointer elements for a given memory allocation header element in response to multiple pointer update messages received from the main processor. The coprocessor may further update previously created memory pointer elements in response to pointer update messages that reference storage addresses of previously created pointers in the main memory. As such, memory pointer elements may be both created and updated with a new value using pointer update messages sent to the coprocessor from the main processor.

To manage the relationships between the created memory allocation header elements and the one or more created memory pointer elements in the second memory, the coprocessor may create and utilize information (e.g., metadata) that relates the various created memory elements within the second memory. This metadata may relate the memory allocation header element previously created for the memory allocation and the created or updated one or more memory pointer elements associated with each memory allocation header element.

It should be noted that a one-to-many relationship may exist between header elements and memory pointer elements. This relationship may occur, for example, when a main processor creates a complex data structure that includes multiple pointers. Conversely, a memory pointer element maps to only one header element. As such, for example, the coprocessor may create a tree structure, such as a trie structure or some other kind of associative container or array (each alternative generally referred to herein as an "associative mapping structure" for ease of description), to identify created memory pointer elements based upon use of the base address and memory allocation size represented within a memory allocation header element. Conversely, the coprocessor may create either a hash table or a tree structure, such as a trie structure or some other kind of associative mapping structure, to identify a memory allocation header element based upon the address represented within memory pointer elements. As such, with a reference to either a memory allocation header element or a memory pointer element, the associated memory pointer element(s) or memory allocation header element, respectively, may be identified within the second memory.

Garbage collection algorithms, such as the various mark and sweep or counting garbage collection algorithms, may be executed by the coprocessor against the memory allocations within the second memory. These various garbage collection algorithms will not be described herein for brevity. However, it is understood that a person of skill in the art will be able to implement such a garbage collection algorithm in association with the present subject matter based upon the description herein.

The hardware off-load garbage collection acceleration for languages that include finalizers described herein may be performed in real time to allow prompt memory garbage collection with reduced primary processor overhead. For purposes of the present description, real time shall include any time frame of sufficiently short duration as to provide reasonable response time for information processing acceptable to a user of the subject matter described. Additionally, the term "real time" shall include what is commonly termed "near real time"—generally meaning any time frame of sufficiently short duration as to provide reasonable response time for on-demand information processing acceptable to a user of the subject matter described (e.g., within a portion of a second or within a few seconds). These terms, while difficult to precisely define are well understood by those skilled in the art.

FIG. **1** is a block diagram of an example of an implementation of a system **100** for automated hardware off-load garbage collection acceleration for languages that include finalizers. A main processor **102** and a coprocessor **104** communicate, as described above and in more detail below, to carry out the automated hardware off-load garbage collection acceleration for languages that include finalizers described herein.

The main processor **102** is operatively coupled via an interconnection **106** to a uni-directional message queue **108** (hereinafter "message queue **108**") to send messages to the coprocessor **104**. Similarly, the coprocessor **104** is operatively coupled via an interconnection **110** to a uni-directional message queue **112** (hereinafter "message queue **112**") to send messages to the main processor **102**. The main processor **102** may be implemented to have only write access to the message queue **108** and only read access to the message queue **112**. Similarly, the coprocessor **104** may be implemented to have only write access to the message queue **112** and only read access to the message queue **108**. The respective message queues may be memory mapped, register mapped, or otherwise accessible for write and read operations, respectively, by the respective processors. Details of implementation for mapping write operations and read operations to the respective message queues are omitted for brevity. However, it is understood that a person of skill in the art would be able to implement such a mapping for the respective processors based upon the description herein, such as by use of read and write signals associated with the respective processors.

The main processor **102** is also operatively coupled via the interconnection **106** with a main memory **114** to create and access memory allocations for use by applications executed by the main processor **102**. The coprocessor **104** is similarly operatively coupled via the interconnection **110** with a coprocessor memory **116** to create and access memory allocations for performing garbage collection activities for the main processor **102**.

It is understood that the main memory **114** and the coprocessor memory **116** may include any combination of volatile and non-volatile memory suitable for the intended purpose, distributed or localized as appropriate, and may include other memory segments not illustrated within the present example for ease of illustration purposes. For example, the main memory **114** and the coprocessor memory **116** may include a code storage area, a code execution area, and a data area without departure from the scope of the present subject matter. As such, code space for run-time memory allocations by the main processor **102** and coprocessor **104** will be described in detail herein and references herein to direct mapping and reduced mapping refer to run-time memory allocation areas, such as data areas, associated with the respective processors.

The interconnection **106** and the interconnection **110** may include a system bus, a network, or any other interconnection capable of providing the respective components with suitable interconnection for the respective purpose.

It should further be noted that the message queue **108** may be sized sufficiently to allow the coprocessor **104** to perform garbage collection without the message queue **108** filling to capacity. This size may be small or zero, if the garbage collection algorithm is one of the concurrent algorithms, such as reference counting. The size may be large if the garbage collection algorithm is not one of the concurrent algorithms, such as mark and sweep. Additionally, for an implementation of either of the interconnection **106** or the interconnection **110** that includes a network or other protocol-based interconnection with associated protocol overhead, the message queue **108** may be sized sufficiently to accommodate interconnection delays for message propagation. Similarly, the message queue **112** may be sized sufficiently to allow the coprocessor **104** to communicate with the main processor **102** and for the main processor **102** to retrieve all messages without the message queue **112** filling to capacity. It should be further noted that software and hardware may maintain its own non-shared queues in addition to these shared queues. If a shared queue is full, the non-shared queues may be used for additional buffering.

As will be described in more detail below in association with FIG. **2** through FIG. **9**, the coprocessor **104** provides automated hardware off-load garbage collection acceleration for languages that include finalizers. The automated hardware off-load garbage collection acceleration for languages that include finalizers is based upon messaging received from the main processor **102** that identifies memory allocations, pointer initialization, and update activities carried out by the main processor **102**. The coprocessor **104** performs garbage collection activities, in association with the memory allocations in the coprocessor memory **116**, to identify memory allocations in the main memory **114** that are no longer in use and that may be freed by the main processor **102**. The coprocessor **104** sends messages to the main processor **102** instructing to the main processor **102** to call finalizer routines for memory allocations that are no longer in use when the allocations have associated finalizers. The coprocessor **104** sends messages to the main processor **102** instructing the main processor **102** to free the unused memory allocations when the allocation does not have a finalizer or in response to receiving messaging from the main processor **102** indicating that the associated finalizer routine has been called. Performance for the main processor **102** may be improved by use of automated hardware off-load garbage collection acceleration for languages that include finalizers.

FIG. **2** described below shows an example message flow for messaging between the main processor **102** and the coprocessor **104**. FIG. **3** described below shows example message formats that may be used for communication between the main processor **102** and the coprocessor **104**. FIG. **4** and FIG. **5** show examples of memory allocation elements used by the direct mapping and reduced mapping implementations, respectively. FIG. **6** shows an example memory organization for the reduced mapping implementation. FIG. **7** through FIG. **9** show example flow charts that may be used to implement the present subject matter. Reference may be made between the respective figures throughout the description below to facilitate further understanding of the present subject matter.

FIG. **2** is a message flow diagram of an example of an implementation of a messaging interaction **200** between the main processor **102** and the coprocessor **104** for automated hardware off-load garbage collection acceleration for languages that include finalizers. For ease of illustration, the message queue **108** and the message queue **112** are omitted from FIG. **2**. However, it is understood that the message queue **108** and the message queue **112**, or a similar messaging system, convey the respective messages to the respective processors, as described in more detail below.

At block **202**, the main processor **102** allocates a memory element within the main memory **114**. The memory allocation performed at block **202** may include allocation of a single byte, an integer, a word, or another storage allocation. The allocation performed at block **202** may further include allocation of a complex data structure, such as an array, an array of pointers, or other complex data structure.

In response to the memory allocation at block **202**, the main processor **102** sends a create object message to the coprocessor **104** (line **1**), including an indication of whether the associated memory allocation has an associated finalizer routine. In response to receipt of the create object message, the coprocessor **104** creates an object representative of the allocated memory element referenced within the create object message at block **204** including the indication of whether the associated memory allocation has an associated finalizer routine.

For purposes of illustration, the present example assumes that the allocated memory at block **202** includes at least one pointer. However, it is understood that the present subject matter may be utilized with non-pointer allocations. It is further assumed, for purposes of the present example, that all of the memory allocations have an associated finalizer routine to be processed by the main processor **102** prior to freeing the respective memory allocation(s). However, it is further understood that the present subject matter may be utilized with memory allocations that do not all have finalizer routines.

At block **206**, the main processor **102** initializes a pointer within the previously-allocated memory area. In response to initialization of the pointer, the main processor **102** sends a pointer update message to the coprocessor **104** (line **2**). The pointer update message includes the address of the previously-allocated memory area where the pointer is initialized and the initialized pointer value (e.g., the address pointed to by the pointer). For a complex data structure within which the pointer is initialized, the address of the pointer that is initialized may be different from a base address of the memory allocation for the complex data structure, as described in more detail below. At block **208**, in response to receipt of the pointer update message, the coprocessor **104** writes the address representative of the pointer value to the object created in block **204**.

At some time later, as represented by the first pair of jagged markings in FIG. **2**, the present example assumes that the main processor **102** changes the pointer value at block **210**. In response to changing the pointer value, the main processor **102** sends another pointer update message to the coprocessor **104** (line **3**). As such, and as can be seen from the present example, the main processor **102** sends a pointer update message in response to both initialization of a pointer value and for changes to pointer values. In response to receipt of the second pointer update message, the coprocessor **104** determines whether an address associated the pointer update message references the object that was previously created (e.g., at block **204**) and initialized (e.g., written at block **208**). In response to determining that the second pointer update message references the object that was previously created and initialized, the coprocessor **104** overwrites the pointer value at block **212**.

Immediately, or at some later time, as represented by the second pair of jagged markings in FIG. **2**, the coprocessor **104** runs a garbage collection algorithm at block **214**. As described above and in more detail below, the garbage collection algorithm may include any garbage collection algorithm appropriate for a given implementation. During the garbage collection activities associated with block **214**, the coprocessor **104** identifies memory objects within the coprocessor memory **116** that represent objects within the main memory **114** that are no longer in use by the main processor **102**.

In response to completion of the garbage collection activities at block **214**, the coprocessor **104** sends a call finalizer message to the main processor **102** for each object that is no longer in use by the main processor **102** (line **4**). As described above, it is assumed for purposes of illustration within the present example, that all memory allocation have an associated finalizer routine. As such, at least one call finalizer message is sent to the main processor **102** for each object to be freed. However, as noted above, it is further understood that the present subject matter may be utilized with memory allocations that do not all have finalizer routines.

The main processor **102** calls all finalizers referenced by the received call finalizer messages at block **216** and sends a finalizer called message to the coprocessor **104** after each finalizer is called (line **5**).

In response to receipt of the finalizer called message for each memory allocation that has an associated finalizer or in response to determining that a particular memory allocation does not have an associated finalizer routine, the coprocessor **104** sends a free object message to the main processor **102** for each object that is no longer in use by the main processor **102** (line **6**). In response to receipt of each free object message, at block **218**, the main processor **102** frees the memory object referenced by each respective free object message within the main memory **114**. It should further be noted that while the present example shows free object messages for memory allocations that do have associated finalizer routines after completion of processing for any memory allocations that have finalizers, this example sequence is for ease of illustration purposes only. Free object messages for memory allocations that do have associated finalizer routines may be sent to the main processor **102** concurrently or before the call finalizer messages without departure from the scope of the present subject matter.

As such, the example messaging interaction **200** between the main processor **102** and the coprocessor **104** allows the main processor **102** to allocate memory without having to run garbage collection to identify memory elements that are no longer used by the main processor **102**. The coprocessor **104** operates as a hardware off-load garbage collection acceleration for languages that include finalizers module and performs garbage collection on behalf of the main processor **102**. The coprocessor **104** instructs the main processor **102** to call finalizer routines for any memory allocations that have associated finalizers. The main processor **102** calls any associated finalizer routines and informs the coprocessor that the finalizer routines have been called. The main processor **102** receives free object messages from the coprocessor **104** and frees objects that have been identified as no longer in use.

Accordingly, the main processor **102** may operate more efficiently and may continue other processing activities while the coprocessor **104** performs garbage collection for the main processor **102**. As discussed above, the message queue **108** and the message queue **112** may be sized as appropriate for a given implementation to accommodate messages that accumulate during garbage collection activities. Further, idle cycles for the main processor **102** may be used to retrieve call finalizer messages and free object messages, and to free the actual memory allocations within the main memory **114**.

FIG. **3** is an illustration of example implementations of message formats that may be used for the messages described in association with FIG. **2** above. A create object message format **300** includes an address of object field **302**, a size of object field **304**, and an address of pointer field **306**. The address of object field **302**, the size of object field **304**, and the address of pointer field **306** include N bits (e.g., labeled zero to N-1). The N bits represent the addressing used on the interconnection **106** by the main processor **102** to access the respective memory object. As such, the number of bits (e.g., N) may be selected based upon the addressing capabilities of a processor for a given implementation.

It should be noted that the address of pointer field **306** represents an address of a pointer value that will receive the address of the allocated memory object as a pointer value. The address of pointer may be used to create a reference to the memory object created within the coprocessor memory **116** to mitigate a race condition between the received create object message and garbage collection processing. Processing to create initial references to memory objects is described in more detail below. For purposes of the present portion of this description, initial references to newly created memory objects may be initialized in association with create object messages to ensure that newly-created memory objects are not freed during garbage collection processing before they are otherwise referenced within the main memory **114** by the main processor **102**.

It should further be noted that alternative processing may be performed by the coprocessor **104** to avoid the messaging overhead associated with adding the address of pointer field **306** to create object messages. The coprocessor **104** may alternatively assign a pointer address value to created memory objects to prevent freeing newly-created memory objects in association with the race condition described above. As another alternative, a requirement may be implemented for the main processor **102** to send a pointer update message immediately after a sending create object message to alleviate the race condition issue while preserving a smaller message format size. Many other alternatives are possible for managing race conditions between the object initialization processing and the garbage collection processing and all are considered within the scope of the present subject matter.

The create object message format **300** further includes a message type field **308**, a root set indicator field **310**, and a finalizer field **312**. The message type field **308** may include, for example, a three-bit message type field. Within the present example, a message type code of "000" binary may be used to indicate that the message was formed using the create object message format **300**. The root set indicator field **310** may include a single bit field and may be used to indicate whether the memory object associated with the create object message format **300** is a root set object, as otherwise described herein. The finalizer field **312** may include a single bit field and may be used to indicate whether the memory allocation has an associated finalizer routine.

A pointer update message format **314** includes an address of pointer field **306** identical to that described above. The pointer update message format **314** further includes a pointer value field **316**. As such, the pointer update message format **314** may be used to indicate, as described above, an address of a pointer and a value to be associated with that pointer for both newly-created pointers and for updated pointers. The pointer update message format **314** also includes the message type field **308** described above. The message type field **308** may include, for purposes of the present example, a value of "001" binary to represent that the message was formed using the pointer update message format **314**.

A call finalizer message format **318** includes an address of object field **302** identical to that described above. However, it is noted that the call finalizer message is sent from the coprocessor **104** to the main processor **102**. As such, a call finalizer message formed using the call finalizer message format **318** represents an instruction to the main processor **102** to call a finalizer associated with a memory object located at the address associated with the call finalizer message. The call finalizer message format **318** also includes the message type field **308** described above. The message type field **308** may include, for purposes of the present example, a value of "010" binary to represent that the message was formed using the call finalizer message format **318**.

A finalizer called message format **320** includes an address of object field **302** identical to that described above. However, it is noted that the finalizer called message is sent from the main processor **102** to the coprocessor **104**. As such, a finalizer called message formed using the finalizer called message format **320** represents a response to the coprocessor **104** that the finalizer associated with a memory object located at the address associated with the call finalizer message has been called. It should be noted that the main processor may autonomously call an object's finalizer and may send the finalizer called message even without a corresponding call finalizer message. The finalizer called message format **320** also includes the message type field **308** described above. The message type field **308** may include, for purposes of the present example, a value of "011" binary to represent that the message was formed using the finalizer called message format **320**.

A free object message format **322** also includes the address of object field **302** identical to that described above. However, it is noted that the free object message is sent from the coprocessor **104** to the main processor **102**. As such, a free object message formed using the free object message format **322** represents an instruction to the main processor **102** to free a memory object located at the address associated with the free object message. The free object message format **322** also includes the message type field **308** as described above. The message type field **308** may include, for purposes of the present example, a value of "100" binary to represent that the associated message was formed using the free object message format **322**.

It should additionally be noted that the respective message formats described above may be considered data structures that may be used to create the associated messages described above. As such, the respective message formats are stored to memory, such as memory associated with the message queue **108** and the message queue **112**, when they are transmitted between the respective processing devices.

FIG. **4** is a block diagram of an example of an implementation of memory allocations for direct mapping between the main memory **114** and the coprocessor memory **116**. A main memory object **400** represents a memory object that is allocated by the main processor **102** within the main memory **114**. As can be seen from FIG. **4**, the main memory object **400** includes a header segment **402** and a body segment **404**. The body segment **404** includes storage for allocated memory associated with the main memory object **400**. The header segment **402** further includes a reserved segment **406** and a header segment **408**. The reserved segment **406** represents an area set aside for use by the coprocessor **104** that is not used by the main processor **102**. The header segment **408** represents a header for use by the main processor **102** for creating and managing the main memory object **400**. The header segment **402** further includes a finalizer indicator **410** that stores an indication of whether the main memory object has an associated finalizer routine to be called prior to freeing the memory allocation.

For purposes of the present description, it is assumed that a header object, such as the header segment **402**, will be associated with each memory object allocated by a processor, such as the main processor **102**, and that additional space will be available for the reserved segment **406**. However, for an implementation that does not allocate a header object, the reserved segment **406** may be established as appropriate for a given implementation.

A coprocessor memory object **412** represents a memory object that is allocated by the coprocessor **104** within the coprocessor memory **116** in response to receipt of a memory allocation message from the main processor **102**. The coprocessor memory object **412** includes a header segment **414** and a body segment **404**. The body segment **404** is identical to the body segment **404** within the main memory object **400**. The header segment **414** further includes a coprocessor management data segment **416** and a header segment **408**. The header segment **408** may be identical to the header segment **408** of the main memory object **400**. The coprocessor management data segment **416** is used by the coprocessor **104** to manage memory allocations for garbage collection, as described above and in more detail below.

The coprocessor management data segment **416** may include, for example, storage space for four bits or flags that may be used for main processor **102** memory allocation tracking and for garbage collection purposes. A root set bit **418** may be used to indicate whether the body segment **404** represents a root set memory object (e.g., static or global variables or pointers, stack variables or pointers, or other root set memory objects). A mark bit **420** may be used to indicate whether the object has been marked by the garbage collection algorithm used in the particular implementation to indicate that the object is still in use.

It is noted that garbage collection algorithms often use a mark counter that inverts for each pass of the respective garbage collection algorithm. As such, the mark bit will be inverted during a mark phase of a mark and sweep garbage collection algorithm to indicate that the object is still in use and will not be inverted if the object is no longer used. Accordingly, the mark bit **418** may be used to mark the coprocessor memory object **410** (and thereby the associated main memory object **400**) for freeing, as described above and in more detail below.

A finalizer status bit field **422** within the coprocessor management data segment **416** may be used to store finalizer status for the memory element for the coprocessor **104**. For example, the finalizer status bit field **422** may represent the following states within the present example using two bits. A setting of "00" binary in the finalizer status bit field **422** may be used to indicate that the memory object has no associated finalizer. A setting of "01" binary in the finalizer status bit field **422** may be used to indicate that the memory object has an associated finalizer, but that the call finalizer message has not been sent to the main processor **102**. A setting of "10" binary in the finalizer status bit field **422** may be used to indicate that the call finalizer message has been sent to the main processor **102**. A setting of "11" binary in the finalizer status bit field **422** may be used to indicate that the finalizer called message has been received from the main processor **102**.

It is further understood that the present example represents one possible implementation of the present organization of the main memory object **400** and the coprocessor memory object **410**. Many other possibilities exist for coprocessor management data, reserved storage relative to the main processor **102**, and other control information. Accordingly, all such possibilities are considered within the scope of the present subject matter. It is also noted that complex data structures may be managed by the coprocessor **104** without additional overhead processing because each memory allocation made by the main processor **102** within the main memory **114** has a matching memory allocation within the coprocessor memory **116** that is made and managed by the coprocessor **104**.

FIG. **5** is a block diagram of an example of an implementation of memory allocations for reduced mapping within the coprocessor memory **116** relative to memory allocations within the main memory **114**. As described above, a reduced mapping implementation may allow for a smaller memory device to be used by the coprocessor **104** relative to a size of the memory device used by the main processor **102**.

A main memory object **500** represents a memory object that is allocated by the main processor **102** within the main memory **114**. As can be seen from FIG. **5**, the main memory object **500** is similar to the main memory object **400** of FIG. **4**. However, the main memory object **500** does not include a reserved area, such as the reserved segment **406** as described above in association with FIG. **4**.

The main memory object **500** includes a header **502** and a body segment **504**. The body segment **504** includes storage for allocated memory associated with the main memory object **500**. The header segment **502** further represents a header for use by the main processor **102** for creating and accessing the main memory object **500**, though it is understood that the header segment **502** may not be necessary for a given implementation of the present subject matter. The header segment **502** further includes a finalizer indicator **506** that stores an indication of whether the main memory object has an associated finalizer routine to be called prior to freeing the memory allocation.

A coprocessor memory allocation header element **512** represents a first portion of a memory object that is allocated by the coprocessor **104** within the coprocessor memory **116** in response to receipt of a memory allocation message from the main processor **102**. The memory allocation header element **512** includes an address segment **514** and a size segment **516**. The address segment **514** includes the address of the main memory object referenced by the body segment **504** within the main memory object **500**. The size segment **516** includes a size of the main memory object **500**. As such, complex data structures may be initially represented by the address represented within the address segment **514** in conjunction with the size represented within the size segment **516**. As will be described in more detail below, data structures that do not include pointers may be referenced by a header, such as the coprocessor memory allocation header element **512**, while pointer allocations and updates are managed by use of a separate memory allocation by the coprocessor **104**.

Coprocessor management data within the coprocessor memory allocation header element **512** includes storage space for two bits or flags that may be used for for garbage collection purposes. A root set bit **518** may be used to indicate whether a memory allocation represented within the address segment **514** and the size segment **516** represents a root set memory object (e.g., static or global variables or pointers, stack variables or pointers, or other root set memory objects). A mark bit **520** may be used to indicate whether the object has been marked by the garbage collection algorithm used in the particular implementation to indicate that the object is still in use.

Finalizer management may be performed using a finalizer status bit field **522** within the coprocessor memory allocation header element **512**, and may be used to store finalizer status for the memory element for the coprocessor **104**. For example, the finalizer status bit field **522** may represent the following states within the present example using two bits. A setting of "00" binary in the finalizer status bit field **522** may be used to indicate that the memory object has no associated finalizer. A setting of "01" binary in the finalizer status bit field **522** may be used to indicate that the memory object has an associated finalizer, but that the call finalizer message has not been sent to the main processor **102**. A setting of "10" binary in the finalizer status bit field **522** may be used to indicate that the call finalizer message has been sent to the main processor **102**. A setting of "11" binary in the finalizer status bit field **522** may be used to indicate that the finalizer called message has been received from the main processor **102**.

Regarding pointer creation, as described above and in more detail below, when the main processor **102** creates a pointer within an allocated memory element, the main processor **102** sends a pointer update message. Upon receipt of a pointer update message, the coprocessor **104** determines whether a memory pointer element has already been created for the pointer referenced by the pointer update message. When a pointer memory element has not been created, the coprocessor **104** creates a memory pointer element. When a pointer memory element has already been created, the coprocessor **104** updates the memory pointer element. In either situation, the value of the pointer (e.g., the pointer's referenced address) is stored within the memory pointer element.

A memory pointer element **524** represents an example memory pointer element that may be created or updated in response to such a pointer update message. The memory pointer element **524** includes an address value segment **526** that stores an address value received from the main processor **102** in association with a pointer update message. As such, the memory pointer element **524** represents an actual pointer value.

Accordingly, the memory pointer element **524** may be considered a value element that may be used in association with a key element, such as used with a key-value pairing in database processing. As such, a key derived from an associated coprocessor memory allocation header element, such as the coprocessor memory allocation header element **512**, may be used to determine the storage location of the memory pointer element **524**. Similarly, a key derived from a memory pointer element may be used to access an associated coprocessor memory allocation header element. As described in more detail below, the respective key derivation may be performed using metadata associated with the respective memory elements.

FIG. **6** is a block diagram of an example of an implementation of memory allocations and associated metadata for reduced mapping within the coprocessor memory **116**. A first coprocessor memory allocation header element **602** (Header_1) represents a first memory allocation received from the main processor **102**. An "Nth" coprocessor memory allocation header element **604** (Header_N) represents a last memory allocation received from the main processor **102**.

A first memory pointer element **606** (Body Object_**1**) represents storage for a first pointer update message (e.g., a pointer initialization) received from the main processor **102**. An "Mth" memory pointer element **608** (Body Object_M) represents storage for a last pointer update message (also a pointer initialization) received from the main processor **102**. It should be noted that the quantity "M" of memory pointer elements may be a different number than the quantity "N" of header elements.

Pointer metadata storage area **610** represents storage for information used for accessing body objects using information stored within a given header element. Similarly, header metadata storage area **612** represents storage for information used for accessing header elements using information stored within a given body object.

It should be noted that a one-to-many relationship may exist between header elements, such as the coprocessor memory allocation header element **602**, and memory pointer elements. This relationship may occur, for example, when the main processor **102** creates a complex data structure that includes multiple pointers. Conversely, a memory pointer element maps to only one header element.

As such, the information stored within the pointer metadata storage area **610** may include, for example, a tree structure, such as a trie structure or some other kind of associative mapping structure, for deriving a key with which to index the body object(s) associated with the respective header element. Conversely, the information stored within the header metadata storage area **612** may include, for example, a hash table or a tree structure, such as a trie structure or some other kind of associative mapping structure, for deriving a key with which to index the header element associated with each body object.

FIG. **7** through FIG. **9** below describe example processes that may be executed by devices, such as the coprocessor **104**, to perform the automated hardware off-load garbage collection acceleration for languages that include finalizers associated with the present subject matter. Many other variations on the example processes are possible and all are considered within the scope of the present subject matter. It should be noted that time out procedures and other error control procedures are not illustrated within the example processes described below for ease of illustration purposes. However, it is understood that all such procedures are considered to be within the scope of the present subject matter.

FIG. **7** is a flow chart of an example of an implementation of a process **700** for automated hardware off-load garbage collection acceleration for languages that include finalizers. At block **702**, the process **700** receives, at a hardware memory management module, a memory allocation message for each primary memory allocation in a primary memory made by a primary processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation. At block **704**, the process **700** allocates, within a second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine. At block **706**, the process **700** determines, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the primary memory. At block **708**, the process **700** sends, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the primary processor instructing the primary processor to call the finalizer routine associated with the primary memory allocation in the primary memory.

FIG. **8** is a flow chart of an example of an implementation of a process **800** for automated hardware off-load garbage collection acceleration for languages that include finalizers to receive and process messages for memory element creation and pointer updating. At decision point **802**, the process **800** makes a determination as to whether a create object message has been received by the coprocessor **104** from the main processor **102**. When a determination is made at decision point **802** that a create object message has not been received, the process **800** makes a determination at decision point **804** as to whether a pointer update message has been received. When a determination is made at decision point **804** that a pointer update message has not been received, the process **800** returns to decision point **802** and iterates as described above.

As also described above, a create object message and a pointer update message may be received at a hardware memory management module, such as the coprocessor **104**, for each primary memory allocation or update in a primary memory made by a primary processor, respectively. Each message may be received from the primary processor via a uni-directional message queue, such as the message queue **108**.

Returning to the description of decision point **802**, when a determination is made that a create object message has been received, the process **800** allocates a memory allocation header at block **806**. As described above, for a direct memory mapping configuration, the coprocessor **104** may allocate, for each primary memory allocation in the primary memory made by the primary processor, the memory element within the coprocessor memory **116** identical to each respective primary memory allocation in the primary memory made by the primary processor. Alternatively, for a reduced memory mapping configuration, the coprocessor **104** may allocate a memory allocation header element within the coprocessor memory **116** that includes a base address and a size of the primary memory allocation within the primary memory. Additionally, as described above, the memory allocation header element may further include an indication that the element represents a root set element, a garbage collection management indicator, and a finalizer status bit field.

At block **808**, the process **800** creates metadata for the header. This header metadata may be used to locate the header using the associated base address, as described above. The process **800** then proceeds to decision point **804**.

Returning to the description of decision point **804**, when the process **800** determines that a pointer update message has been received, the process **800** retrieves the header metadata associated with the pointer address received in the pointer update message at block **810**. At block **812**, the process **800** identifies the header element based upon the retrieved header metadata. At decision point **814**, the process **800** makes a determination as to whether there is any existing pointer metadata associated with the identified header element. When a determination is made at decision point **814** that there is no pointer metadata associated with the identified header element, such as for a pointer initialization operation, the process **800** allocates a memory pointer element at block **816**. As described above, the pointer update message may include a primary memory address associated with a primary memory allocation and a pointer value. As such, at block **818**, the process **800** initializes the memory pointer element with the received pointer value. At block **820**, the process **800** creates pointer metadata to locate/index the memory pointer element from the base address in the header element. As such, the process **800** indexes the memory pointer element relative to the memory allocation header element.

It is further understood, that for complex data structures that may include multiple pointers, multiple pointer elements may be associated with a single header element. The element size, as described above in association with the header format, may be used to determine whether multiple pointer elements may be associated with a given header element. It should further be noted that indexing the memory pointer element relative to the memory allocation header element may further include generating pointer metadata that associate the memory allocation header element with one or more memory pointer elements. In such an implementation, the pointer metadata may identify a storage location of the memory pointer element within the coprocessor memory **116** derived via a tree structure, such as a trie structure or some other kind of associative mapping structure, based upon information within the memory allocation header element.

At block **822**, the process **800** creates header metadata to locate/index the header element from the pointer value and the memory pointer element. The process **800** may generate header metadata that associates the memory pointer element with the memory allocation header element. The header metadata may identify a storage location of the memory allocation header element within the coprocessor memory **116** derived via a hash table, or a tree structure, such as a trie structure or some other kind of associative mapping structure, based upon information within the memory pointer element. The process **800** then returns to decision point **802** and iterates as described above.

Returning to the description of decision point **814**, when the process **800** determines that pointer metadata does already exist, the process **800** also determines that the pointer element, associated with the pointer update message, has already been created in the coprocessor memory **116**. As such, at block **824**, the process **800** retrieves the pointer metadata that was previously created and associated with the identified header element. At block **826**, the process **800** identifies the memory pointer element. At block **828**, the process **800** updates the value of the memory pointer element. At block **830**, the process **800** updates the pointer metadata and header metadata based upon the new pointer value stored in the memory pointer element. The process **800** then returns to decision **802** and continues iterating as described above.

As such, the process **800** provides for creation of memory elements through allocation of memory header elements and header metadata. The process **800** also provides for creation, initialization, and updating of pointers by allocating pointer elements and creating pointer metadata to reference created pointer elements back to the respective header elements. In response to receipt of create object messages and pointer update messages, the process **800** creates, manages, and updates the header metadata and the pointer metadata associated with the respective memory elements.

FIG. **9** is a flow chart of an example of an implementation of a process **900** for automated hardware off-load garbage collection acceleration for languages that include finalizers using a processor memory stack to instruct a main processor, such as the main processor **102**, to free memory allocations. At decision point **902**, the process **900** makes a determination as to whether to perform a garbage collection activity. When a determination is made not to perform a garbage collection activity, the process **900** waits for an indication to begin garbage collection. As described above, garbage collection may be scheduled periodically, non-periodically, or incrementally as appropriate for a given implementation. Further, it should be noted that the garbage collection processing described in association with the process **900** is used to determine, based upon allocated representations of primary memory allocations within the coprocessor memory **116**, whether to free the respective primary memory allocations. It should further be noted that the process **900** may be executed concurrently with the main processor **102** continuing to create memory allocations within the main memory **114**. As described above, the message queue **108** may be sized to accommodate enough messages to allow the process **900** to execute without loss of any memory allocation messages.

The process **900** allows a primary processor, such as the main processor **102**, to continue memory allocations in a primary memory, such as the main memory **114**, concurrently with the hardware off-load module determining to free primary memory allocations in the primary memory that are represented within a coprocessor memory, such as the coprocessor memory **116**. As such, additional received memory allocation messages sent by the main processor **102** during a period of time that the hardware off-load module determines to free the primary memory allocations in the main memory **114** may be processed by a process, such as the process **800** associated with FIG. **8** above, after completion of execution of the process **900**. It should further be noted that execution of the process **800** and the process **900** may be interleaved such that the process **800** and the process **900** also run concurrently.

Returning to the description of decision point **902**, when a determination is made to begin garbage collection activities, the process **900** makes a determination at decision point **904** as to whether all root-set header elements have been processed. It is noted that the first iteration of the process **900** will result in a negative determination if at least one root-set header element has been allocated previously. It should further be noted that the processing associated with the determination at decision point **904** is performed with respect to root-set headers. As such, when a determination is made at decision point **904** that all header elements have not been processed, the process **900** retrieves a first header element at block **906**. At block **908**, the process **900** marks the header. As described above, garbage collection management data associated with the respective header element, such as the marked indication, may be used to mark the header. At block **910**, the process **900** retrieves pointer metadata associated with the retrieved header element.

At block **912**, the process **900** gets the next pointer associated with the retrieved header. At decision point **914**, the process **900** makes a determination as to whether all pointers associated with the retrieved header have been processed. When a determination is made that all pointers associated with the retrieved header have not been processed, the process **900** retrieves the header associated with the respective pointer at block **916** using the header metadata associated with the pointer that is being processed. At decision point **918**, the process **900** makes a determination as to whether the header is already marked. It should be noted that the retrieved header will not be marked during the first iteration of the process **900**, though the retrieved header may already be marked in association with processing during other iterations of the process **900**.

When a determination is made at decision point **918** that the header is already marked, the process **900** returns to block **912** and gets the next pointer associated with the header retrieved at block **906** and iterates as described above. When a determination is made at decision point **918** that the header is not already marked, the process **900** saves/pushes the next pointer location onto the stack at block **920** and returns to block **908** to mark the header and iterates as described above.

Returning to the description of decision point **914**, when a determination is made that all pointers associated with the retrieved header have been processed, the process **900** makes a determination at decision point **922** as to whether the stack is empty. When a determination is made that the stack is not empty, the process **900** populates the next pointer location at block **924**, returns to block **912** to get the next pointer and iterates as described above. When a determination is made that the stack is empty at decision point **922**, the process **900** returns to decision point **904** to determine whether all root-set headers have been processed. As such, the process **900** recursively processes all root-set headers and all pointers with header references until all root-set headers have been processed. When a determination is made at decision point **904** that all root-set headers have been processed, the process **900** identifies unused headers (e.g., headers that have not been marked) at block **926**. At block **928**, the process **900** sends a call object finalizer message to the main processor **102** for each object with an associated finalizer. At decision point **930**, the process **900** makes a determination as to whether all anticipated finalizer called messages have been received from the main processor **102**. When a determination is made that all finalizer called messages have been received, the process **900** sends a free object message to the main processor **102** for each identified unused element at block **932**. The process **900** then returns to decision point **902** to await a new garbage collection indication.

It should be noted that the process **900** shows processing at block **928** and decision point **930** prior to the processing at block **932** for ease of description purposes. It should be understood that free object messages for any objects without finalizers may be sent concurrently with the call object finalizer messages at block **928** without departure from the scope of the present subject matter.

As such, the process **900** performs garbage collection activities on behalf of the main processor **102** using the processor memory stack. The process **900** processes allocated header elements and associated pointer metadata to identify all pointers associated with each header element. Each pointer is processed to identify its associated header metadata to further identify additional header elements associated with the respective memory allocations. As such, the process **900** recursively processes root-set headers using pointer metadata and pointers using header metadata, respectively, to perform a mark phase of a garbage collection algorithm. The process **900** also performs the sweep phase to identify all unused elements that were not marked during the mark phase and sends a call object finalizer message to the main processor **102** for each object identified as having an associated finalizer. A free object message is sent to the main processor **102** after all finalizer called messages have been received for each unused element that has a finalizer. The processing described above may be performed, for example, by computing a transitive closure of a reachability graph for allocated objects. Objects that are not found to be part of the transitive closure may be freed since they are not referenced.

As described above in association with FIG. **1** through FIG. **9**, the example systems and processes provide hardware off-load garbage collection acceleration for languages that include finalizers, such as Java® or other garbage collected languages that include finalizers. Many other variations and additional activities associated with hardware off-load garbage collection acceleration for languages that include finalizers, such as Java® or other garbage collected languages that include finalizers, are possible and all are considered within the scope of the present subject matter.

Those skilled in the art will recognize, upon consideration of the above teachings, that certain of the above examples are based upon use of a programmed processor, such as the coprocessor **104**. However, the invention is not limited to such example embodiments, since other embodiments could be implemented using hardware component equivalents such as special purpose hardware and/or dedicated processors. Similarly, general purpose computers, microprocessor based computers, micro-controllers, optical computers, analog computers, dedicated processors, application specific circuits and/or dedicated hard wired logic may be used to construct alternative equivalent embodiments.

As will be appreciated by one skilled in the art, aspects of the present invention may be embodied as a system, method or computer program product. Accordingly, aspects of the present invention may take the form of an entirely hardware embodiment, an entirely software embodiment (including firmware, resident software, micro-code, etc.) or an embodiment combining software and hardware aspects that may all generally be referred to herein as a "circuit," "module" or "system." Furthermore, aspects of the present invention may take the form of a computer program product embodied in one or more computer readable medium(s) having computer readable program code embodied thereon.

Any combination of one or more computer readable medium(s) may be utilized. The computer readable medium may be a computer readable signal medium or a computer readable storage medium. A computer readable storage medium may be, for example, but not limited to, an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system, apparatus, or device, or any suitable combination of the foregoing. More specific examples (a non-exhaustive list) of the computer readable storage medium would include the following: an electrical connection having one or more wires, a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a portable compact disc read-only memory (CD-ROM), an optical storage device, a magnetic storage device, or any suitable combination of the foregoing. In the context of this document, a computer readable storage medium may be any tangible medium that can contain, or store a program for use by or in connection with an instruction execution system, apparatus, or device.

A computer readable signal medium may include a propagated data signal with computer readable program code embodied therein, for example, in baseband or as part of a carrier wave. Such a propagated signal may take any of a variety of forms, including, but not limited to, electro-magnetic, optical, or any suitable combination thereof. A computer readable signal medium may be any computer readable medium that is not a computer readable storage medium and that can communicate, propagate, or transport a program for use by or in connection with an instruction execution system, apparatus, or device.

Program code embodied on a computer readable medium may be transmitted using any appropriate medium, including but not limited to wireless, wireline, optical fiber cable, RF, etc., or any suitable combination of the foregoing.

Computer program code for carrying out operations for aspects of the present invention may be written in any combination of one or more programming languages, including an object oriented programming language such as Java, Smalltalk, C++ or the like and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The program code may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider).

Aspects of the present invention are described below with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems) and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer program instructions. These computer program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

These computer program instructions may also be stored in a computer-readable storage medium that can direct a computer or other programmable data processing apparatus to function in a particular manner, such that the instructions stored in the computer-readable storage medium produce an article of manufacture including instructions which implement the function/act specified in the flowchart and/or block diagram block or blocks.

The computer program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other devices to cause a series of operational steps to be performed on the computer, other programmable apparatus or other devices to produce a computer implemented process such that the instructions which execute on the computer or other programmable apparatus provide processes for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks.

The flowchart and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods and computer program products according to various embodiments of the present invention. In this regard, each block in the flowchart or block diagrams may represent a module, segment, or portion of code, which comprises one or more executable instructions for implementing the specified logical function(s). It should also be noted that, in some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts, or combinations of special purpose hardware and computer instructions.

A data processing system suitable for storing and/or executing program code will include at least one processor coupled directly or indirectly to memory elements through a system bus. The memory elements can include local memory employed during actual execution of the program code, bulk storage, and cache memories which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

Input/output or I/O devices (including but not limited to keyboards, displays, pointing devices, etc.) can be coupled to the system either directly or through intervening I/O controllers.

Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems and Ethernet cards are just a few of the currently available types of network adapters.

The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting of the invention. As used herein, the singular forms "a", "an" and "the" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms "comprises" and/or "comprising," when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof.

The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of the present invention has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art without departing from the scope and spirit of the invention. The embodiment was chosen and described in order to best explain the principles of the invention and the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

What is claimed is:

1. A method, comprising:

receiving, at a hardware memory management module, a memory allocation message for each primary memory allocation in a primary memory made by a primary processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation;

allocating, within a second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine;

determining, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the primary memory; and

sending, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the primary processor instructing the primary processor to call the finalizer routine associated with the primary memory allocation in the primary memory.

2. The method of claim 1, further comprising:

receiving a finalizer called message from the primary processor; and

sending a memory free message to the primary processor instructing the primary processor to free the primary memory allocation in the primary memory in response to receiving the finalizer called message.

3. The method of claim 1, further comprising:

sending, in response to determining that the primary memory allocation does not comprise the associated finalizer routine, a memory free message to the primary processor instructing the primary processor to free the primary memory allocation.

4. The method of claim 1, where the second memory comprises a memory device identical in size to the primary memory, and where allocating, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine comprises:

allocating, for each primary memory allocation in the primary memory made by the primary processor, a memory element within the second memory identical to each respective primary memory allocation in the primary memory made by the primary processor; and

setting a finalizer field associated with each memory element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

5. The method of claim 1, where the second memory comprises a memory device smaller in size than the primary memory, and where allocating, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine comprises:

allocating, for each primary memory allocation in the primary memory made by the primary processor, a memory allocation header element within the second memory comprising a base address and a size of the primary memory allocation within the primary memory; and

setting a finalizer field associated with each memory allocation header element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

6. The method of claim 1, further comprising:

receiving a memory pointer update message from the primary processor in response to a pointer initialization performed by the primary processor, where the memory pointer update message comprises a primary memory address associated with a primary memory allocation and a pointer value;

identifying a memory allocation header element with a base address that matches the primary memory address;

allocating a memory pointer element within the second memory;

indexing the memory pointer element relative to the memory allocation header element; and

initializing the memory pointer element with the pointer value.

7. The method of claim 1, where determining, based upon the allocated representations of each primary memory allocation within the second memory, to free the primary memory allocation in the primary memory comprises:

executing one of a memory reference counting garbage collection algorithm and a mark and sweep garbage collection algorithm against the allocated representations of each primary memory allocation within the second memory; and

determining, based upon the one of the memory reference counting garbage collection algorithm and the mark and sweep garbage collection algorithm executed against the allocated representations of each primary memory allocation within the second memory, to free the primary memory allocation in the primary memory.

8. A system, comprising:

a first processor operatively coupled to a first memory;

a bi-directional message queue; and

an off-load processor operatively coupled to a second memory, and programmed to:

receive, via the bi-directional message queue, a memory allocation message for each primary memory allocation in the first memory made by the first processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation;

allocate, within the second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine;

determine, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the first memory; and

send, via the bi-directional message queue, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the first processor instructing the first processor to call the finalizer routine associated with the primary memory allocation in the first memory.

9. The system of claim 8, where the off-load processor is further programmed to:

receive a finalizer called message from the primary processor; and

send a memory free message to the primary processor instructing the primary processor to free the primary memory allocation in the primary memory in response to receiving the finalizer called message.

10. The system of claim 8, where the off-load processor is further programmed to:

send, in response to determining that the primary memory allocation does not comprise the associated finalizer routine, a memory free message to the primary processor instructing the primary processor to free the primary memory allocation.

11. The system of claim 8, where the second memory comprises a memory device identical in size to the first memory and where, in being programmed to allocate, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine, the off-load processor is programmed to:

allocate, for each primary memory allocation in the first memory made by the first processor, a memory element within the second memory identical to each respective primary memory allocation in the first memory made by the first processor; and

set a finalizer field associated with each memory element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

12. The system of claim **8**, where the second memory comprises a memory device smaller in size than the first memory and where, in being programmed to allocate, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine, the off-load processor is programmed to:

allocate, for each primary memory allocation in the first memory made by the first processor, a memory allocation header element within the second memory comprising a base address and a size of the primary memory allocation within the first memory; and

set a finalizer field associated with each memory allocation header element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

13. The system of claim **8**, where the off-load processor is further programmed to:

receive a memory pointer update message from the first processor in response to a pointer initialization performed by the first processor, where the memory pointer update message comprises a primary memory address associated with a primary memory allocation and a pointer value;

identify a memory allocation header element with a base address that matches the primary memory address;

allocate a memory pointer element within the second memory;

index the memory pointer element relative to the memory allocation header element; and

initialize the memory pointer element with the pointer value.

14. A computer program product comprising a computer readable storage device including computer readable program code, wherein the computer readable program code when executed on a computer causes the computer to:

receive a memory allocation message for each primary memory allocation in a first memory made by a first processor, where each memory allocation message comprises an indication of whether a finalizer routine is associated with each primary memory allocation;

allocate, within a second memory in response to each memory allocation message, a representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine;

determine, based upon the allocated representations of each primary memory allocation within the second memory, to free a primary memory allocation in the first memory; and

send, in response to determining that the primary memory allocation comprises the associated finalizer routine, a call object finalizer message to the first processor instructing the first processor to call the finalizer routine associated with the primary memory allocation in the first memory.

15. The computer program product of claim **14**, where the computer readable program code when executed on the computer further causes the computer to:

receive a finalizer called message from the primary processor; and

send a memory free message to the primary processor instructing the primary processor to free the primary memory allocation in the primary memory in response to receiving the finalizer called message.

16. The computer program product of claim **14**, where the computer readable program code when executed on the computer further causes the computer to:

send, in response to determining that the primary memory allocation does not comprise the associated finalizer routine, a memory free message to the primary processor instructing the primary processor to free the primary memory allocation.

17. The computer program product of claim **14**, where the second memory comprises a memory device identical in size to the first memory and where, in causing the computer to allocate, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine, the computer readable program code when executed on the computer causes the computer to:

allocate, for each primary memory allocation in the first memory made by the first processor, a memory element within the second memory identical to each respective primary memory allocation in the first memory made by the first processor; and

set a finalizer field associated with each memory element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

18. The computer program product of claim **14**, where the second memory comprises a memory device smaller in size than the first memory and where, in causing the computer to allocate, within the second memory in response to each memory allocation message, the representation of each primary memory allocation comprising the indication of whether each primary memory allocation comprises the associated finalizer routine, the computer readable program code when executed on the computer causes the computer to:

allocate, for each primary memory allocation in the first memory made by the first processor, a memory allocation header element within the second memory comprising a base address and a size of the primary memory allocation within the first memory; and

set a finalizer field associated with each memory allocation header element based upon the indication of whether the finalizer routine is associated with each primary memory allocation.

19. The computer program product of claim **14**, where the computer readable program code when executed on the computer further causes the computer to:

receive a memory pointer update message from the first processor in response to a pointer initialization performed by the first processor, where the memory pointer update message comprises a primary memory address associated with a primary memory allocation and a pointer value;

identify a memory allocation header element with a base address that matches the primary memory address;

allocate a memory pointer element within the second memory;

index the memory pointer element relative to the memory allocation header element; and

initialize the memory pointer element with the pointer value.

20. The computer program product of claim **14**, where, in causing the computer to determine, based upon the allocated representations of each primary memory allocation within the second memory, to free the primary memory allocation in the first memory, the computer readable program code when executed on the computer causes the computer to:

execute one of a memory reference counting garbage collection algorithm and a mark and sweep garbage collection algorithm against the allocated representations of each primary memory allocation within the second memory; and

determine, based upon the one of the memory reference counting garbage collection algorithm and the mark and sweep garbage collection algorithm executed against the allocated representations of each primary memory allocation within the second memory, to free the primary memory allocation in the first memory.

* * * * *