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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 June 2017 (08.06.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/095367 A1

(51) International Patent Classification:
G06F 12/02 (2006.01) *G06F 12/00* (2006.01)

(21) International Application Number:
PCT/US2015/062931

(22) International Filing Date:
30 November 2015 (30.11.2015)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HEWLETT PACKARD ENTERPRISE DEVELOPMENT LP** [US/US]; 11445 Compaq Center Drive West, Houston, Texas 77070 (US).

(72) Inventor: **KIRSHENBAUM, Evan R.**; 1501 Page Mill Rd., Palo Alto, California 94304-1100 (US).

(74) Agents: **D'ADDARIO, Daniel M.** et al.; Hewlett Packard Enterprise, 3404 E. Harmony Road, Mail Stop 79, Fort Collins, Colorado 80528 (US).

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

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

Published:

— *with international search report (Art. 21(3))*

(54) Title: MANAGING OBJECTS STORED IN MEMORY

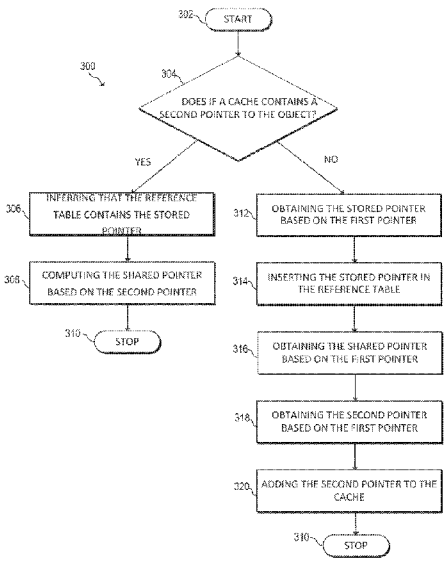

FIG. 3

(57) Abstract: In one example in accordance with the present disclosure, a method may include receiving a first pointer to an object and ensuring that a stored pointer to the object exists in a reference table. The method may also include obtaining a shared pointer to the object, the shared pointer having an associated reference count and modifying the reference count to indicate a number of a number of copies of the shared pointer within a computer system. The method may also include determining that the reference count indicates that there are no remaining copies of the shared pointer in the computer system and removing the stored pointer from the reference table upon making the determination.

WO 2017/095367 A1

MANAGING OBJECTS STORED IN MEMORY

Background

[0001] Garbage collection is a type of memory management where a garbage collector reclaims memory occupied by objects that are no longer in use. Garbage collection may be used and/or required by certain programming languages. Although garbage collection may provide significant benefits, garbage collection may require some system overhead and thus may impact performance.

Brief Description of the Drawings

[0002] The following detailed description references the drawings, wherein:

[0003] FIG. 1 is a block diagram of an example computing environment in which managing objects stored in memory may be useful;

[0004] FIG. 2 is a flowchart of an example method for obtaining a shared pointer to the object referred to by the first pointer;

[0005] FIG. 3 is a flowchart of an example method for identifying pointers in a cache;

[0006] FIG. 4 is a flowchart of an example method for managing objects stored in memory;

[0007] FIG. 5 is a block diagram of an example system for managing objects stored in memory; and

[0008] FIG. 6 is a block diagram of an example system for managing objects stored in memory.

Detailed Description

[0009] The data accessed within a computer system may comprise a set of objects (also known as records, structures, instances, or simply data values) stored in the memory or storage of the computer system. Access to objects may be by use of pointers (or references or links) that contain sufficient information to identify the object they refer to. This information may include an address of the object, an offset of a common base address, a key that can be looked up in a table, or other information. These pointers may be stored within processor registers, on a program

stack, in global variables, within objects, and elsewhere. In a system that employs garbage collection (GC), some objects may be allocated on a heap managed by a garbage collector. When an object is allocated, unused space is identified on the heap, and the object is created in that space. The job of the garbage collector is to distinguish between allocated objects that are reachable and those that are unreachable, where an object is reachable if and only if it is possible for any program code to make reference to the object. When objects are determined to be unreachable, the garbage collector declares the space they occupy to be unallocated and returns the memory to the allocator for use in allocating new objects.

[0010] At the beginning of a GC cycle, a garbage collector may determine all of the root pointers for each process that is using the heap. A root pointer is a pointer to an object stored in a garbage-collected heap where the root pointer itself is observed by a garbage collector in a location other than on the garbage-collected heap. For example, the pointer may be observed in a processor register, on a program or thread stack, in a global or thread-local variable or structure, or on a heap other than the garbage-collected heap).

[0011] The objects pointed to by the root pointers are considered reachable, as are all objects pointed to by GC pointers contained within reachable objects. Any part of the GC-managed heap that is not reachable may be considered garbage and used in future object allocation. Root pointers to objects on GC heap can be found on a thread's local stack. To find root pointers on the thread's local stack, the threads may be scanned and root pointers may be identified. There are, however, numerous other places where root pointers to objects on the GC heap can be found, such as in global variables, class-static variables, function-static variables, objects created on the process's local heap, etc. The system for managing objects provides a centralized way to manage and update values of root pointers that exists on a shared heap.

[0012] Example systems for managing objects stored in memory discussed herein manage storage using reference counting. A number, referred to as a reference count, may be used to indicate a number of pointers to an object. A pointer reflected in an object's reference count may be called reference counted (or counting) pointers (or references) or shared pointers (or references). One example shared pointer that may be used is the standard C++ shared_ptr<T> class, which

implements a shared pointer to objects of a type T. When shared pointers are copied or assigned, the resulting pointer may be associated with the same reference count and that count is incremented. When shared pointers are destroyed or overwritten by an assignment, the associated reference count is decremented. A reference count being decremented to zero is indicative of there being no further shared pointers referring to the object, and the object may be deleted and its space made available for future object allocation.

[0013] The reference count may be maintained in a separate object, called a control block. The lifetime of a control block may be independent of the lifetime of the object, and the reference counting pointer may contain the address (or equivalent location information) of both the object and the control block associated with the object. Example systems for managing objects stored in memory discussed herein may also contain weak reference counted (or counting) pointers (or references), also known as weak pointers (or references). One example of a weak pointer that may be used is the standard C++ weak_ptr<T> class. Weak pointers may refer to an object and an associated reference count (or control block), but the existence of weak pointers is not reflected in the associated reference count. Thus weak pointers can continue to exist even when their associated reference counts are zero. A shared pointer may be made based on a weak pointer. This shared pointer may refer to the same object the weak pointer referred if the weak pointer referred to an object (e.g., was not a null pointer) and its associated reference count was not zero before the creation of the shared pointer. Otherwise, the shared pointer will be a null pointer. To manage the lifetime of the control block object, the control block may contain a separate counter reflecting the number of weak pointers associated with the control block. The control block may also contain information (e.g., data and references to machine instructions) sufficient to delete the object and make its space available for future allocation.

[0014] Example systems for managing objects stored in memory may remove a stored pointer corresponding to the object from a reference table when the reference count has reached zero, as opposed to automatically deleting the object pointed to by the stored pointer. A stored pointer is a pointer in a reference table used to manage shared pointers. Accordingly, multiple shared pointers can point to the same object and each shared pointer may have its own reference count. Because

the object is not deleted when the reference counter reaches zero, the shared pointer corresponding to the reference counter that reached zero is removed from the reference table without affecting the underlying object or any other shared pointers that point to the object. An example method may include may include receiving a first pointer to an object and ensuring that a stored pointer to the object exists in a reference table. The method may also include obtaining a shared pointer to the object, the shared pointer having an associated reference count and modifying the reference count to indicate a number of a number of copies of the shared pointer within a computer system. The method may also include determining that the reference count indicates that there are no remaining copies of the shared pointer in the computer system and removing the stored pointer from the reference table upon making the determination.

[0015] FIG. 1 is a block diagram of an example system 100 for managing objects stored in memory of a computer system. System 100 may include a processor 102 and a memory 104 that may be coupled to each other through communication link (e.g., a bus). Processor 102 may include a Central Processing Unit (CPU) or another suitable processor. In some examples, memory 104 stores machine readable instructions executed by processor 102 for system 100. Memory 104 may include any suitable combination of volatile and/or non-volatile memory, such as combinations of Random Access Memory (RAM), Read-Only Memory (ROM), flash memory, and/or other suitable memory.

[0016] Memory 104 stores instructions to be executed by processor 102 including instructions for a pointer receiver 112, a pointer ensurer 114, a pointer obtainer 116, a reference count modifier 118, a reference count handler 120, a pointer remover 122, a cache handler 124, a pointer enumerator 126, and/or other components. According to various implementations, system 100 for managing objects may be implemented in hardware and/or a combination of hardware and programming that configures hardware. Furthermore, in FIG. 1 and other Figures described herein, different numbers of components or entities than depicted may be used.

[0017] Processor 102 may execute instructions of pointer receiver 112 to receive a first pointer to an object. The first pointer may be, for example, a garbage collection pointer. The garbage collection pointer may be an instance of a GC pointer type (e.g., a class of an object oriented language). The garbage collection pointer may represent

a pointer to an object of type T that exists on a shared heap managed by the garbage collector. The heap may be contained within a non-volatile memory.

[0018] Processor 102 may execute instructions of pointer ensurer 114 to ensure that a stored pointer to the object exists in a reference table. A stored pointer is a pointer in a reference table used to manage shared pointers (e.g. as discussed below in reference to pointer obtainer 116). A stored pointer may be created for each non-stack reference to a GC managed object. When a garbage collection pointer is created, a corresponding stored pointer may be created, including an associated shared pointer, and the stored pointer may be put in the reference table.

[0019] Pointer ensurer 114 may identify a slot in the reference table as an unused slot and remove the slot from a list of unused slots. Pointer ensurer 114 may further construct the stored pointer based on the first pointer and store the stored pointer in the slot in the reference table.

[0020] The reference table may be used to manage the root pointers for a process. For example, the reference table may be an array of garbage collection pointers that holds root pointers pointing to objects stored in the GC managed heap, where the objects are reachable by code running in the process via references other than those on the stack of any of the threads of the process. The reference table may be a multi-level table. For example, the reference table may be implemented as a two-level array comprising a fixed-size array of pointers to lazily-constructed fixed size arrays of slots. The upper array may be referred to as the spine and the lower arrays may be referred to as blocks. One example reference table contains block each containing 10,000 slots and a spine containing 100,000 block pointers, allowing for a maximum of one billion active root pointers. This example reference table is only an example and other reference tables may have different numbers of spines, blocks and root pointers. A slot may contain a root pointer. When a slot is an empty slot (e.g., it contains a null pointer rather than a root pointer), it may contain an index or other indication of another empty slot or an indication that it does not refer to another empty slot. In this way, the reference table may contain one or more free lists of empty slots. The reference table may also be associated with an indication of the slot with the greatest index that has ever been claimed to receive a root pointer.

[0021] Processor 102 may execute instructions of pointer obtainer 116 to obtain a shared pointer to the object. The shared pointer may have an associated reference

count (e.g., located within a control block object pointed to by the shared pointer and by all copies of the shared pointer). Processor 102 may execute instructions of reference count modifier 118 to modify the reference count to indicate a copies of the shared pointer within the computer system. The copies of the shared pointer may include the shared pointer itself, each copy of the shared pointer, copies of the copies, etc. As the shared pointer and its copies are copied or destroyed, the reference count may be incremented and decremented. Processor 102 may execute instructions of reference count handler 120 to determine that the reference count indicates that there are no remaining copies of the shared pointer in the computer system. The reference count may also be associated (e.g., in the control block) with information sufficient to locate the stored pointer in the reference table. Information sufficient to locate the stored pointer in the reference table may be an index into the reference table, an address or other location, a pointer to the address or other location, etc.

[0022] Processor 102 may execute instructions of pointer remover 122 to remove the pointer from the reference table when the last copy of the shared pointer is destroyed or assigned a different value (i.e., the reference count becomes zero). Pointer remover 122 may remove the pointer from the reference table by identifying the slot containing the pointer by using information associated (e.g., in the control block) and replacing the pointer in the slot with a null pointer. Pointer remover 122 may also add the now-empty slot to a free list of empty slots (e.g., the free list associated with the current thread's thread-local cache, as described below, in regards to Fig. 3). Adding the slot to the free list may comprise copying the indication of the head of the free list into the slot as the next empty slot and setting the head of the free list to refer to the slot.

[0023] Processor 102 may execute instructions of cache handler 124 to determine whether a cache contains a second pointer to the object. This is explained in further detail below, in regards to Fig. 3. When a shared pointer is created from the first pointer, the system may determine if a pointer associated with the first pointer, such as, for example, a copy of the first pointer, already exists in the reference table by checking for pointers referring to the object referred to by the first pointer in a cache. As will be discussed in further detail below in regards to Fig. 3, the cache may include

a thread-local cache and/or a global cache. In this manner, the system may prevent duplicate entries to be entered in the reference table.

[0024] Processor 102 may execute instructions of pointer enumerator 126 to enumerate pointers in the reference table. Pointer enumerator 126 may enumerate each pointer in the reference table. Each object pointed to by an enumerated pointer may be considered to be a live object by a garbage collector managing the heap.

[0025] FIG. 2 is a flowchart of an example method 200 for obtaining a shared pointer to the object referred to by the first pointer. Method 200 may be described below as being executed or performed by a system, for example, system 100 of FIG. 1, system 500 of FIG. 5 or system 600 of FIG. 6. Other suitable systems and/or computing devices may be used as well. Method 200 may be implemented in the form of executable instructions stored on at least one machine-readable storage medium of the system and executed by at least one processor of the system. Alternatively or in addition, method 200 may be implemented in the form of electronic circuitry (e.g., hardware). The steps of method 200 may be executed substantially concurrently or in a different order than shown in FIG. 2. Method 200 may include more or less steps than are shown in FIG. 2. The steps of method 200 may, at certain times, be ongoing and/or may repeat.

[0026] Method 200 may start at step 202 and continue to step 204, where the method may include identifying an empty slot in the reference table. As discussed above, the reference table may be used to manage a set of root pointers. Identifying an empty slot may include identifying a slot as the head of a free list of empty slots (e.g., the free list associated with the current thread's thread-local cache, as described below with regard to Fig. 3). If the free list is empty, identifying an empty slot may further include first obtaining a new free list (e.g., from the collection of free lists associated with the global cache, as described below with regard to Fig. 3, or obtaining a never-before used slot from the reference table). At step 206, the method may include claiming the empty slot in the reference table to prevent the slot from being subsequently identified as an empty slot before it is cleared. Claiming the empty slot may involve removing the slot from the free list of empty slots, as by replacing the head of the free list by the next empty slot indicator associated with the slot. Alternatively, claiming the empty slot may involve setting an indicator (e.g., a bit or flag) associated with the slot. At step 208, the method may include installing a stored pointer (e.g. a GC pointer) into the empty slot. The stored pointer may point to the object pointed to by the first pointer

and at step 210, the method may include creating a shared pointer. The shared pointer may correspond to the first pointer. The shared pointer may point to the same object that the first pointer points to. When the last copy of the shared pointer is destroyed, the stored pointer may be removed from the table. For example, the shared pointer may be associated (e.g., in the control block) with a deleter object that stores the index of the slot and which, when called when the reference count associated with the shared pointer reaches zero, clears the slot (e.g., by overwriting the stored pointer by a null pointer).

[0027] FIG. 3 is a flowchart of an example method 300 for identifying pointers in a cache. Method 300 may be described below as being executed or performed by a system, for example, system 100 of FIG. 1, system 500 of FIG. 5 or system 600 of FIG. 6. Other suitable systems and/or computing devices may be used as well. Method 300 may be implemented in the form of executable instructions stored on at least one machine-readable storage medium of the system and executed by at least one processor of the system. Alternatively or in addition, method 300 may be implemented in the form of electronic circuitry (e.g., hardware). The steps of method 300 may be executed substantially concurrently or in a different order than shown in FIG. 3. Method 300 may include more or less steps than are shown in FIG. 3. The steps of method 300 may, at certain times, be ongoing and/or may repeat.

[0028] Method 300 may start at step 302 and continue to step 304, where the method may include determining if a cache contains a second pointer to the object. The second pointer may be a weak pointer or any form of pointer based on which a shared pointer may be created.

[0029] Determining whether the cache contains the second pointer to the object may also include obtaining a weak pointer associated with the first pointer. The weak pointer may be in the cache. The method may also include obtaining a third pointer based on the weak pointer and determining whether the third pointer points to the object. The third pointer may be a shared pointer or any pointer which can be constructed from a weak pointer. Upon determining that the third pointer points to the object, the method may include identifying the weak pointer as the second pointer or otherwise computing the second pointer based on the weak pointer, including computing the second pointer based on the third pointer.

[0030] If the cache does contain the second pointer, (YES branch of step 304), the method 300 may continue to step 306, where method 300 may include inferring, based on the second pointer being in the cache, that the reference table contains the stored pointer. In other words, if the second pointer is in the cache, it can be inferred that the reference table contains the stored pointer, without checking the reference table itself.

[0031] As discussed above, a shared pointer to the object, with an associated reference count, may be obtained. At step 308, the method may include computing the shared pointer based on the second pointer. For example, the method may include identifying the third pointer as the shared pointer, and the third pointer may be computed based on the weak pointer identified with the second pointer. Method 300 may eventually continue to step 310, where method 300 may stop.

[0032] If the cache does not contain a second pointer to the object, (NO branch of step 304), the method 300 may continue to step 312, where method 300 may include obtaining the stored pointer based on the first pointer (e.g., by creating the stored pointer to point to the object referred to by the first pointer). At step 314, the method may include inserting the stored pointer in the reference table. At step 316, the method may include obtaining the shared pointer based on the first pointer (e.g., by creating the shared pointer to point to the object referred to by the first pointer). Obtaining the shared pointer based on the first pointer may comprise creating a new control block, different from all other control blocks in the system, and associating it with the shared pointer. At step 318, the method may include obtaining the second pointer based on the first pointer. The second pointer may be obtained, for example, by creating a weak pointer copy of the shared pointer, which is obtained based on the first pointer. At step 320 the method may include adding the second pointer to the cache. Method 300 may eventually continue to step 310, where method 300 may stop.

[0033] The cache may comprise two levels and may include a thread-local cache and a global cache. A thread local cache may be created for each program thread using the GC managed heap and may only be accessed by the program thread it was created for. The thread local cache may be created on demand. The thread-local cache may also include an immutable reference to the process's reference table, an immutable reference to the global cache, a fixed-sized array of weak references and an integer or other indication of a first empty slot of the reference table in a free list of empty slots.

[0034] The global cache may be associated with a plurality of threads using the GC managed heap. There may be a single global cache per operating system process containing threads using the GC managed heap. The global cache may include a fixed size array of entries, each of which contains a weak reference and a flag. The flag may be used to ensure that only one thread at a time is accessing an entry in the reference table. An example flag that may be used is an atomic_flag supported by the C++ programming language. The atomic_flag in C++ is an object that supports the *clear( )* and the *test_and_set ( )*operations. The *clear( )* operation atomically sets the atomic flag to false and the *test_and_set( )* operation atomically sets the flag to true and returns the prior value. If the *test_and_set( )* operation on a given atomic_flag object returns a false value, this indicates that the invoking thread was the first to invoke the operation on the object or the first to do so after the *clear( )* operation was invoked on the object. The global cache may further include a collection of free lists of empty slots (e.g., indications of the first slots of such free lists). When thread-local caches are destroyed (e.g., upon thread exit), their associated free lists, when non-empty, may be added to the global cache's collection of free lists.

[0035] When converting the garbage collector pointer (e.g., the first pointer) to the shared pointer, a lookup operation may be performed by a local cache handler in the thread-local cache of the thread. The lookup operation may accept the garbage collector pointer as an argument and return the shared pointer. If the value of garbage collector pointer is null, a null shared pointer may be returned. If the value of the garbage collector pointer is not null, the local cache handler may obtain a bare pointer (e.g., a C++ T* pointer) to the object the GC pointer points to. The local cache handler may then determine the index into the array corresponding to the bare pointer. The index may be determined by converting the bare pointer to a number, performing shift operations on the number to remove any low-order bits known to be zero due to the in-memory alignment of the object pointed to, and taking the modulus of the resulting number with the array size. In example implementations in which the thread-local cache's array size is constrained to be a power of two, this modulus may be performed by means of a bitwise AND operation using the appropriate bit mask. The weak pointer contained at the index in the thread-local-cache's weak pointer array may converted to a shared pointer (e.g., by the C++ weak_ptr<T>::*lock()* function).

[0036] If the value of the resulting shared pointer is null, the array slot may be empty or it may contain a weak pointer to an object where the last shared pointer associated with the weak pointer's reference count was destroyed. If the value of the shared pointer is not equal to the bare pointer (e.g., if it is null or if points to a different object than does the garbage collector pointer), then it can be determined that the thread-local cache does not contain a weak pointer referring to the correct object, since there is only one index that such a weak pointer could reside at. In this scenario, the local cache handler may request the global cache handler to perform a lookup operation in the global cache based on the bare pointer. Specifically, the method 300 may include determining that the thread-local cache does not contain a second pointer (i.e. as discussed above in regards to step 304) and determining that the global cache contains the second pointer. The method may further include adding a value based on the second pointer to the thread-local cache.

[0037] The lookup operation performed by the global cache handler will be discussed in further detail below. The global cache handler may return the result of the lookup operation to the local cache handler as a shared pointer. If the shared pointer received from the global cache handler matches the bare pointer, the local cache handler may put a weak pointer copy of the shared pointer in the thread-local cache's weak pointer array at the index previously computed. The local cache handler may return the shared pointer, which may be either the shared pointer found during the thread-local cache lookup operation or the shared pointer received from the global cache, as the result of the lookup operation. The weak pointers in the thread-local cache and global cache may refer to objects as being of a common type, which may be a base type in a hierarchy of types. In some examples, when the type of object referred to by the garbage collector type passed in as an argument to the thread-local lookup operation differs from that of the shared pointer determined by the local cache handler, the local cache handler may cast the shared pointer to the appropriate shared pointer type (e.g., corresponding to the GC pointer type) and return the resulting shared pointer.

[0038] A lookup operation may be also be performed in the global cache by a global cache handler. The global cache's *lookup* method takes a (bare) pointer to the object as a parameter and a creation function and returns a shared pointer to the object. The creation function may create a stored pointer based on a GC pointer

(e.g., the GC pointer known to the local cache handler) and insert it into the reference table. The creation function may further create a shared pointer pointing to the same object as the GC pointer and associated with a new control block not associated with any other shared pointer. The creation function may return the created shared pointer. The global cache handler may determine (e.g., by means similar to those used by the local cache handler to determine an index) the index into the global cache's array corresponding to the pointer, where the index uniquely identifies an entry in the global cache's array. The global cache handler may then request an exclusive access to the identified entry by attempting to set the entry's flag. For example, the global cache handler may invoke the *test_and_set( )* operation on the atomic_flag corresponding to the entry. An exclusive access means that no entity other than the entity with exclusive access can access the entry. In other words, if the global cache handler has exclusive access to the identified entry, no other entity can access the identified entry. An exclusive may be useful in preventing other entities (i.e. other processes, threads, etc. from accessing the entry during the lookup operation. If the result of this operation is a true value, indicating that the flag was set prior to the invocation, the global cache handler may infer that a global cache handler operating in another thread has acquired exclusive access to the entry. If the result of the operation is false, then the invocation changed the flag from unset to set, and the global cache handler operating in this thread has acquired exclusive access to the entry.

[0039] If exclusive access has not been acquired, the global cache handler may infer that another thread is currently using the entry and that it is therefore unsafe for this thread to do so. In this situation, the global cache handler may invoke the creation function and return the resulting shared pointer. As a side-effect of invoking the creation function, a stored pointer may be created and added to the reference table. If exclusive access has not been acquired, the global cache handler may return from the lookup operation without clearing the entry's flag. If exclusive access to the entry has been acquired, the global cache handler may obtain a shared pointer based on the weak pointer contained in the entry (e.g., by means of the C++ weak_ptr<T>::*lock()* method). If the resulting shared pointer refers to the same object as the pointer parameter, then the global cache contains a pointer (i.e., the weak pointer) to the object, and the global cache handler may relinquish its exclusive

access to the entry (e.g., by calling the *clear()* method on the entry's flag) and return the shared pointer. If the resulting shared pointer does not refer to the same object as the pointer parameter, then the global cache handler may call the creation function to create a new shared pointer referring to the object. As a side-effect of invoking the creation function, a stored pointer may be created and added to the reference table. The global cache handler may create a weak pointer based on the shared pointer and may store the weak pointer in the entry. It may then relinquish its exclusive access to the entry and return the shared pointer. Specifically, the method 300 may include identifying an entry in the global cache associated with the first pointer, requesting an exclusive access to the entry, obtaining a weak pointer associated with the entry and relinquishing the exclusive access

[0040] FIG. 4 is a flowchart of an example method 400 for managing objects stored in the memory of a computer system. Method 400 may be described below as being executed or performed by a system, for example, system 100 of FIG. 1, system 500 of FIG. 5 or system 600 of FIG. 6. Other suitable systems and/or computing devices may be used as well. Method 400 may be implemented in the form of executable instructions stored on at least one machine-readable storage medium of the system and executed by at least one processor of the system. Alternatively or in addition, method 400 may be implemented in the form of electronic circuitry (e.g., hardware). In alternate examples of the present disclosure, at least one step of method 400 may be executed substantially concurrently or in a different order than shown in FIG. 4. In alternate examples of the present disclosure, method 400 may include more or less steps than are shown in FIG. 4. In some examples, at least one of the steps of method 400 may, at certain times, be ongoing and/or may repeat.

[0041] Method 400 may start at step 402 and continue to step 404, where the method may include receiving a first pointer to an object. The first pointer may be, for example, a garbage collection pointer. The garbage collection pointer may belong to a class of an object oriented language. The object may be stored in a GC managed heap. At step 406, the method may include ensuring that a stored pointer to the object exists in a reference table. The reference table may be a multi-level table. Ensuring that the stored pointer to the object exists in a reference table may further include identifying a slot in the reference table as an unused slot, claiming the slot, constructing the stored pointer based on the first pointer and storing the stored pointer in the slot. At step 408,

the method may include obtaining a shared pointer to the object. The shared pointer may have an associated reference count. The reference count may be associated with information sufficient to locate the stored pointer in the reference table. At step 410, the method may include modifying the reference count to indicate a number of copies of the shared pointer within a computer system. The number of instances may include the shared pointer and each copy of the shared pointer. At step 412, the method may include determining that the reference count indicates that there are no remaining copies of the shared pointer in the computer system. At step 414, the method may include removing the stored pointer from the reference table. Step 414 may be performed upon making the determination in step 412.

[0042] FIG. 5 is a block diagram illustrating one example of a processing system 500 for implementing the system 500 for managing objects stored in the memory of a computer system. System 500 may include a processor 502 and a memory 504 that may be coupled to each other through a communication link (e.g., a bus). Processor 502 may include a Central Processing Unit (CPU) or another suitable processor. In some examples, memory 504 stores machine readable instructions executed by processor 502 for system 500. Memory 504 may include any suitable combination of volatile and/or non-volatile memory, such as combinations of Random Access Memory (RAM), Read-Only Memory (ROM), flash memory, and/or other suitable memory.

[0043] Memory 504 stores instructions to be executed by processor 502 including instructions for a pointer receiver 510, cache handler 512, pointer creator 514, reference count modifier 516 and pointer remover 518. The components of system 500 may be implemented in the form of executable instructions stored on at least one machine-readable storage medium of system 500 and executed by at least one processor of system 500. Alternatively or in addition, each of the components of system 500 may be implemented in the form of at least one hardware device including electronic circuitry for implementing the functionality of the component.

[0044] Processor 502 may execute instructions of pointer receiver 502 to receive a first pointer to an object. The first pointer may be, for example, a garbage collection pointer. The garbage collection pointer may belong to a class of an object oriented language. The object may be stored in a GC managed heap. Processor 502 may execute instructions of cache handler 504 to determine that a cache does not contain a second pointer to the object. The cache may include a thread-local cache associated

with a thread within the computer system and a global cache associated with a plurality of threads within the computer system. The plurality of threads may include the thread. Processor 502 may execute instructions of pointer creator 506 to create a stored pointer and a shared pointer corresponding to first pointer. The shared pointer may have an associated reference count. The reference count may be associated with information sufficient to locate the stored pointer in the reference table. Processor 502 may execute instructions of reference count modifier 508 to modify the reference count to indicate a number of instances of the shared pointer within the computer system. The number of instances may include the shared pointer and each copy of the shared pointer. Processor 502 may execute instructions of pointer remover 510 to remove the stored pointer from a reference table when the reference count indicates that there are no remaining instances of the shared pointer in the computer system. The reference table may be a multi-level table.

[0045] FIG. 6 is a block diagram of an example system 600 for managing objects stored in the memory of a computer system. System 600 may be similar to system 100 of FIG. 1, for example. In the example illustrated in FIG. 5, system 600 includes a processor 602 and a machine-readable storage medium 604. Although the following descriptions refer to a single processor and a single machine-readable storage medium, the descriptions may also apply to a system with multiple processors and multiple machine-readable storage mediums. In such examples, the instructions may be distributed (e.g., stored) across multiple machine-readable storage mediums and the instructions may be distributed (e.g., executed by) across multiple processors.

[0046] Processor 602 may be one or more central processing units (CPUs), microprocessors, and/or other hardware devices suitable for retrieval and execution of instructions stored in machine-readable storage medium 604. In the example illustrated in FIG. 6, processor 602 may fetch, decode, and execute instructions 606, 608, 610, 612, 614, 616 and 618 for managing objects stored in the memory of a computer system. As an alternative or in addition to retrieving and executing instructions, processor 602 may include one or more electronic circuits comprising a number of electronic components for performing the functionality of at least one of the instructions in machine-readable storage medium 604. With respect to the executable instruction representations (e.g., boxes) described and shown herein, it should be understood that part or all of the executable instructions and/or electronic circuits included within one box

may, in alternate examples, be included in a different box shown in the figures or in a different box not shown.

[0047] Machine-readable storage medium 604 may be any electronic, magnetic, optical, or other physical storage device that stores executable instructions. Thus, machine-readable storage medium 604 may be, for example, Random Access Memory (RAM), an Electrically-Erasable Programmable Read-Only Memory (EEPROM), a storage drive, an optical disc, and the like. Machine-readable storage medium 604 may be disposed within system 600, as shown in FIG. 6. In this situation, the executable instructions may be "installed" on the system 600. Alternatively, machine-readable storage medium 604 may be a portable, external or remote storage medium, for example, that allows system 600 to download the instructions from the portable/external/remote storage medium. In this situation, the executable instructions may be part of an "installation package". As described herein, machine-readable storage medium 604 may be encoded with executable instructions for using pointers in a memory managed system.

[0048] Referring to FIG. 6, pointer receive instructions 606, when executed by a processor (e.g., 602), may cause system 600 to receive a first pointer. The first pointer may correspond to an object. The object may be stored in a GC managed heap. The first pointer may be, for example, a garbage collection pointer. The garbage collection pointer may belong to a class of an object oriented language. The reference table may be a multi-level table. Local cache instructions 608, when executed by a processor (e.g., 602), may cause system 600 to determine that no pointer corresponding to the first pointer exists in a local cache. The local cache may correspond to a thread using the first pointer. Global cache instructions 610, when executed by a processor (e.g., 602), may cause system 600 to determine that a second pointer corresponding to the first pointer exists in a global cache. The global cache may correspond to a plurality of threads including the first thread sharing a centrally managed data store.

[0049] Pointer create instructions 612, when executed by a processor (e.g., 602), may cause system 600 to create a stored pointer and a shared pointer corresponding to the first pointer, the shared pointer having a reference count. The shared pointer may have a reference count. The reference count may be associated with information sufficient to locate the stored pointer in a reference table. Pointer add instructions 614, when executed by a processor (e.g., 602), may cause system 600 to add the

stored pointer to a reference table. Reference count modify instructions 616, when executed by a processor (e.g., 602), may cause system 600 to modify the reference count to indicate a number of instances of the shared pointer within the computer system. The number of instances may include the shared pointer and each copy of the shared pointer. Pointer remove instructions 618, when executed by a processor (e.g., 602), may cause system 600 to remove the stored pointer from the reference table when the reference count indicates that there are no remaining instances of the shared pointer in the computer system.

[0050] The foregoing disclosure describes a number of examples for managing objects stored in the memory of a computer system. The disclosed examples may include systems, devices, computer-readable storage media, and methods for using pointers in a memory managed system. For purposes of explanation, certain examples are described with reference to the components illustrated in FIGS. 1-6. The functionality of the illustrated components may overlap, however, and may be present in a fewer or greater number of elements and components. Further, all or part of the functionality of illustrated elements may co-exist or be distributed among several geographically dispersed locations. Further, the disclosed examples may be implemented in various environments and are not limited to the illustrated examples.

[0051] Further, the sequence of operations described in connection with FIGS. 1-6 are examples and are not intended to be limiting. Additional or fewer operations or combinations of operations may be used or may vary without departing from the scope of the disclosed examples. Furthermore, implementations consistent with the disclosed examples need not perform the sequence of operations in any particular order. Thus, the present disclosure merely sets forth possible examples of implementations, and many variations and modifications may be made to the described examples.

Claims

1. A method comprising:

receiving a first pointer to an object;

ensuring that a stored pointer to the object exists in a reference table;

obtaining a shared pointer to the object, the shared pointer having an associated reference count;

modifying the reference count to indicate a number of copies of the shared pointer within a computer system;

determining that the reference count indicates that there are no remaining copies of the shared pointer in the computer system; and

removing the stored pointer from the reference table upon making the determination.

2. The method of claim 1, further comprising:

enumerating each pointer in the reference table, wherein each object pointed to by an enumerated pointer is considered to be a live object by a garbage collector.

3. The method of claim 1, wherein the reference count is associated with information sufficient to locate the stored pointer in the reference table.

4. The method of claim 1, further comprising:

determining whether a cache contains a second pointer to the object;

wherein if it is determined that the cache contains the second pointer:

inferring, based on the second pointer being in the cache, that the reference table contains the stored pointer; and

obtaining the shared pointer based on the second pointer;

wherein if it is determined that the cache does not contain the second pointer:

obtaining the stored pointer based on the first pointer;

inserting the stored pointer in the reference table;

obtaining the shared pointer based on the first pointer;

obtaining the second pointer based on the first pointer; and

adding the second pointer to the cache.

5. The method of claim 4, wherein the determining whether the cache contains the second pointer further comprises:

obtaining a weak pointer associated in the cache with the first pointer;

obtaining a third pointer based on the weak pointer;

determining whether the third pointer points to the object; and

upon determining that the third pointer points to the object, compute the second pointer based on the weak pointer.

6. The method of claim 1, wherein the cache comprises a thread-local cache associated with a thread within the computer system and a global cache associated with a plurality of threads within the computer system, wherein the plurality of threads includes the thread, the method further comprising:

determining that the thread-local cache does not contain a second pointer to the object;

determining that the global cache contains the second pointer to the object; and

adding a value based on the second pointer to the thread-local cache.

7. The method of claim 6, further comprising:

identifying an entry in the global cache associated with the first pointer;

requesting an exclusive access to the entry; and

obtaining a weak pointer associated with the entry; and

relinquishing the exclusive access.

8. The method of claim 1, wherein ensuring that the stored pointer to the object exists in the reference table comprises:

identifying a slot in the reference table as an unused slot;

claiming the slot;

constructing the stored pointer based on the first pointer; and

storing the stored pointer in the slot.

9. The method of claim 1, wherein the reference table is a multi-level table.

10. A system comprising:

a pointer receiver to receive a first pointer to an object;

a cache handler to determine that a cache does not contain a second pointer to the object;

a pointer creator to create a stored pointer and a shared pointer corresponding to first pointer, the shared pointer having an associated reference count;

a reference count modifier to modify the reference count to indicate a number of instances of the shared pointer within the computer system, where the number of instances include the shared pointer and each copy of the shared pointer; and

a pointer remover to remove the stored pointer from a reference table when the reference count indicates that there are no remaining instances of the shared pointer in the computer system.

11. The system of claim 10, wherein the reference count is associated with information sufficient to locate the stored pointer in the reference table.

12. The system of claim 10, wherein the cache comprises a thread-local cache associated with a thread within the computer system and a global cache associated with a plurality of threads within the computer system, wherein the plurality of threads includes the thread and the cache handler is further to:

determine that the thread-local cache does not contain a second pointer to the object;

determine that the global cache contains the second pointer to the object; and

add to the thread-local cache a value based on the first pointer.

13. The system of claim 10, further comprising a pointer enumerator to enumerate each pointer in the reference table, wherein each object pointed to by an enumerated pointer is considered to be a live object by a garbage collector.

14. A non-transitory machine-readable storage medium comprising instructions executable by a processor of a computing device, the machine-readable storage medium comprising instructions to:

receive a first pointer corresponding to an object;

determine that no pointer corresponding to the first pointer exists in a local cache, the local cache corresponding to a first thread using the first pointer;

determine that a second pointer corresponding to the first pointer exists in a global cache, the global cache corresponding to a plurality of threads including the first thread sharing a centrally managed data store;

create a stored pointer and a shared pointer corresponding to the first pointer, the shared pointer having a reference count;

add the stored pointer to a reference table;

modify the reference count to indicate a number of instances of the shared pointer within the computer system, where the number of instances include the shared pointer and each copy of the shared pointer; and

remove the stored pointer from the reference table when the reference count indicates that there are no remaining instances of the shared pointer in the computer system.

15. The non-transitory machine-readable storage medium of claim 14 further comprising instructions to:

identify an entry in the global cache associated with the first pointer;

request an exclusive access to the entry;

obtain a weak pointer associated with the entry and;

relinquish the exclusive access.

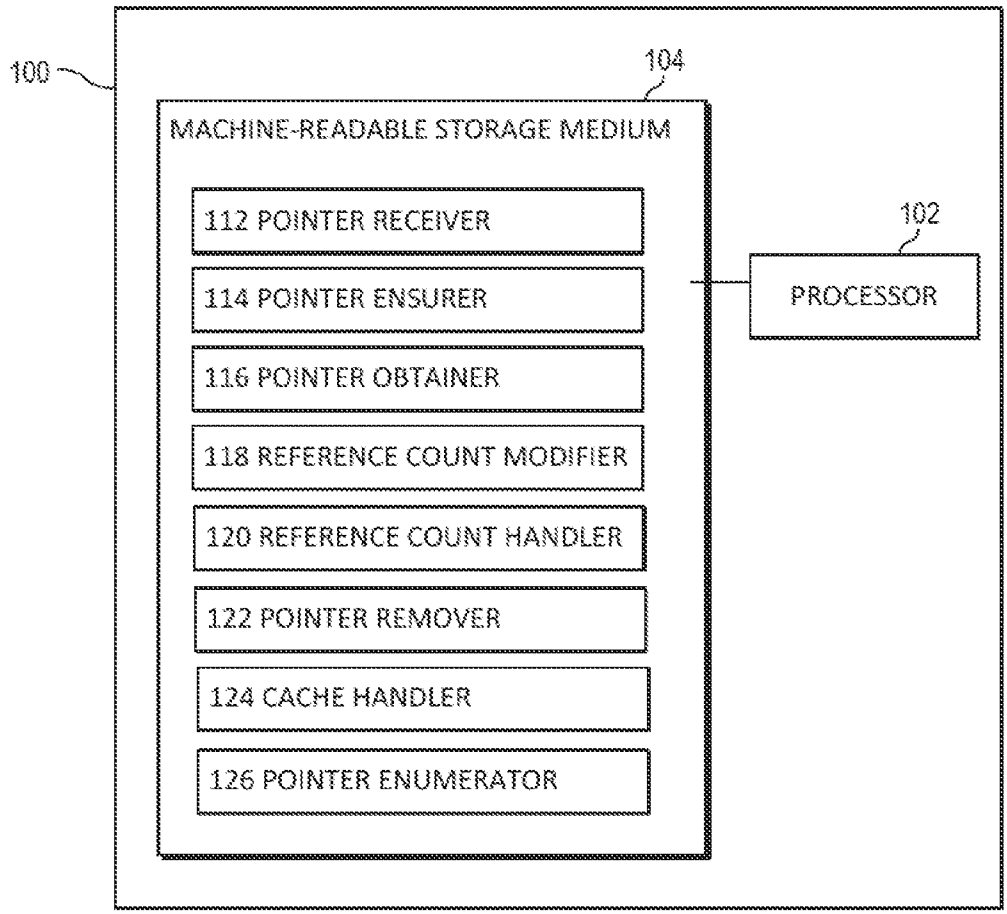
100
104
MACHINE-READABLE STORAGE MEDIUM
112 POINTER RECEIVER
114 POINTER ENSURER
116 POINTER OBTAINER
118 REFERENCE COUNT MODIFIER
120 REFERENCE COUNT HANDLER
122 POINTER REMOVER
124 CACHE HANDLER
126 POINTER ENUMERATOR
102
PROCESSOR

FIG. 1

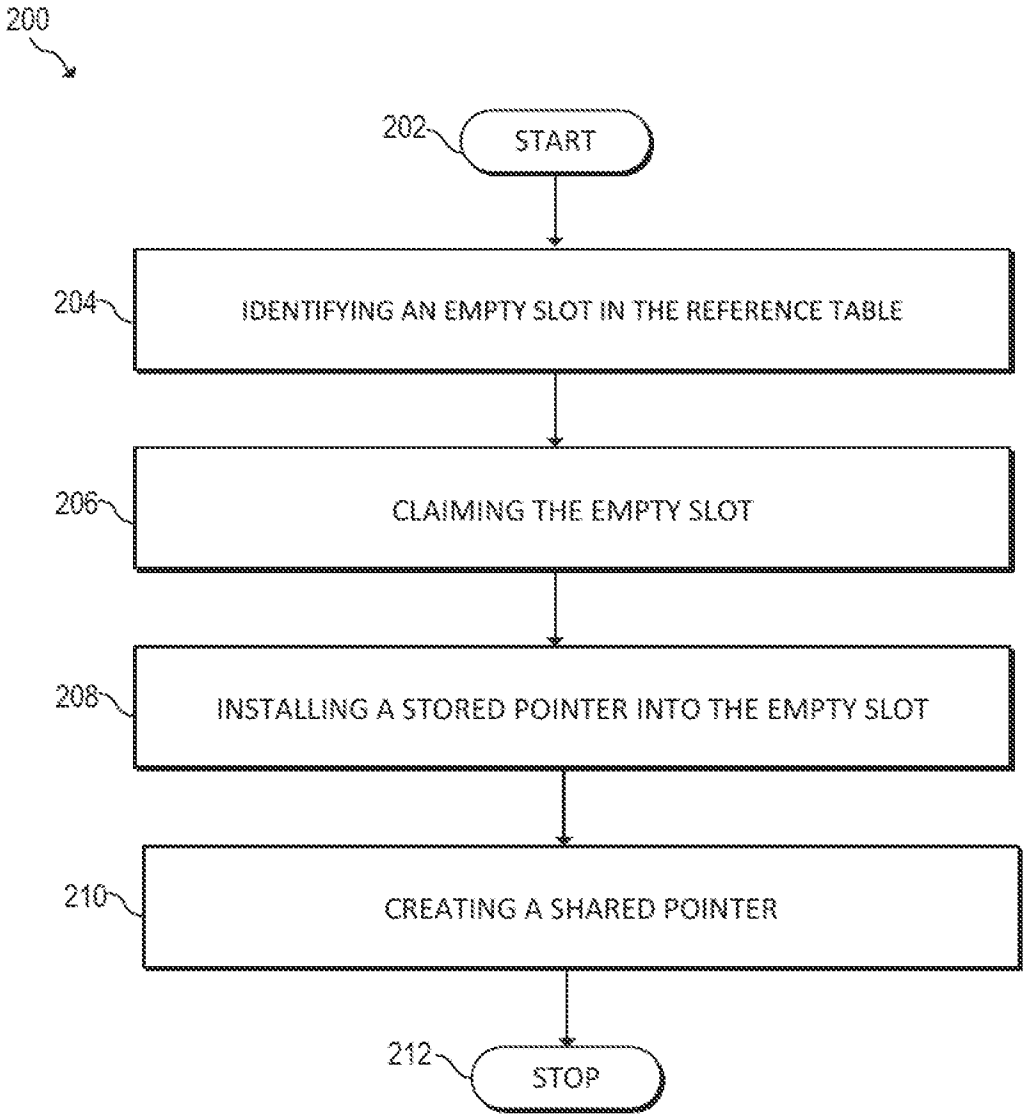
200
202
START
204
IDENTIFYING AN EMPTY SLOT IN THE REFERENCE TABLE
206
CLAIMING THE EMPTY SLOT
208
INSTALLING A STORED POINTER INTO THE EMPTY SLOT
210
CREATING A SHARED POINTER
212
STOP

FIG. 2

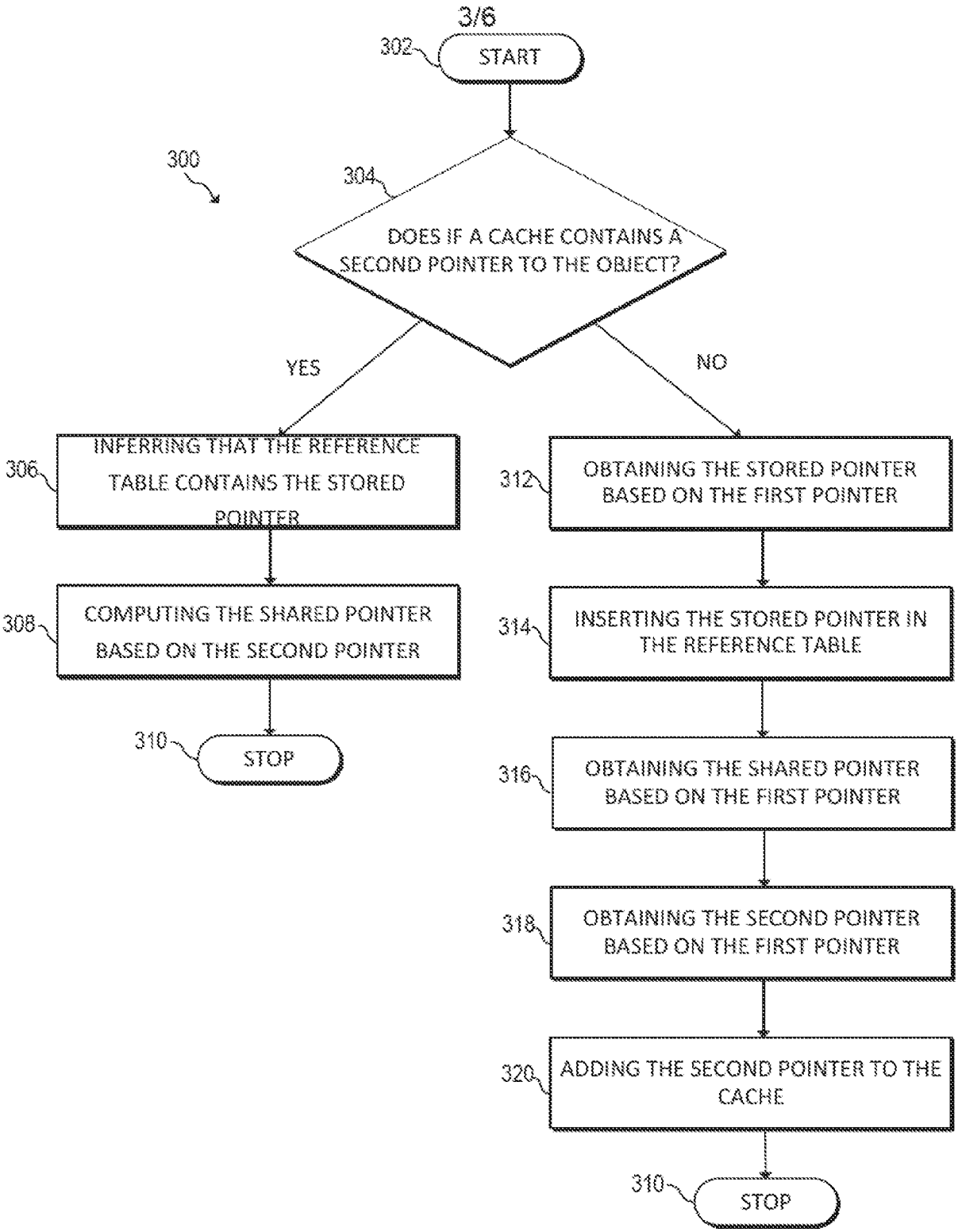
300
302
START
304
DOES IF A CACHE CONTAINS A SECOND POINTER TO THE OBJECT?
YES
NO
306
INFERRING THAT THE REFERENCE TABLE CONTAINS THE STORED POINTER
308
COMPUTING THE SHARED POINTER BASED ON THE SECOND POINTER
310
STOP
312
OBTAINING THE STORED POINTER BASED ON THE FIRST POINTER
314
INSERTING THE STORED POINTER IN THE REFERENCE TABLE
316
OBTAINING THE SHARED POINTER BASED ON THE FIRST POINTER
318
OBTAINING THE SECOND POINTER BASED ON THE FIRST POINTER
320
ADDING THE SECOND POINTER TO THE CACHE
310
STOP

FIG. 3

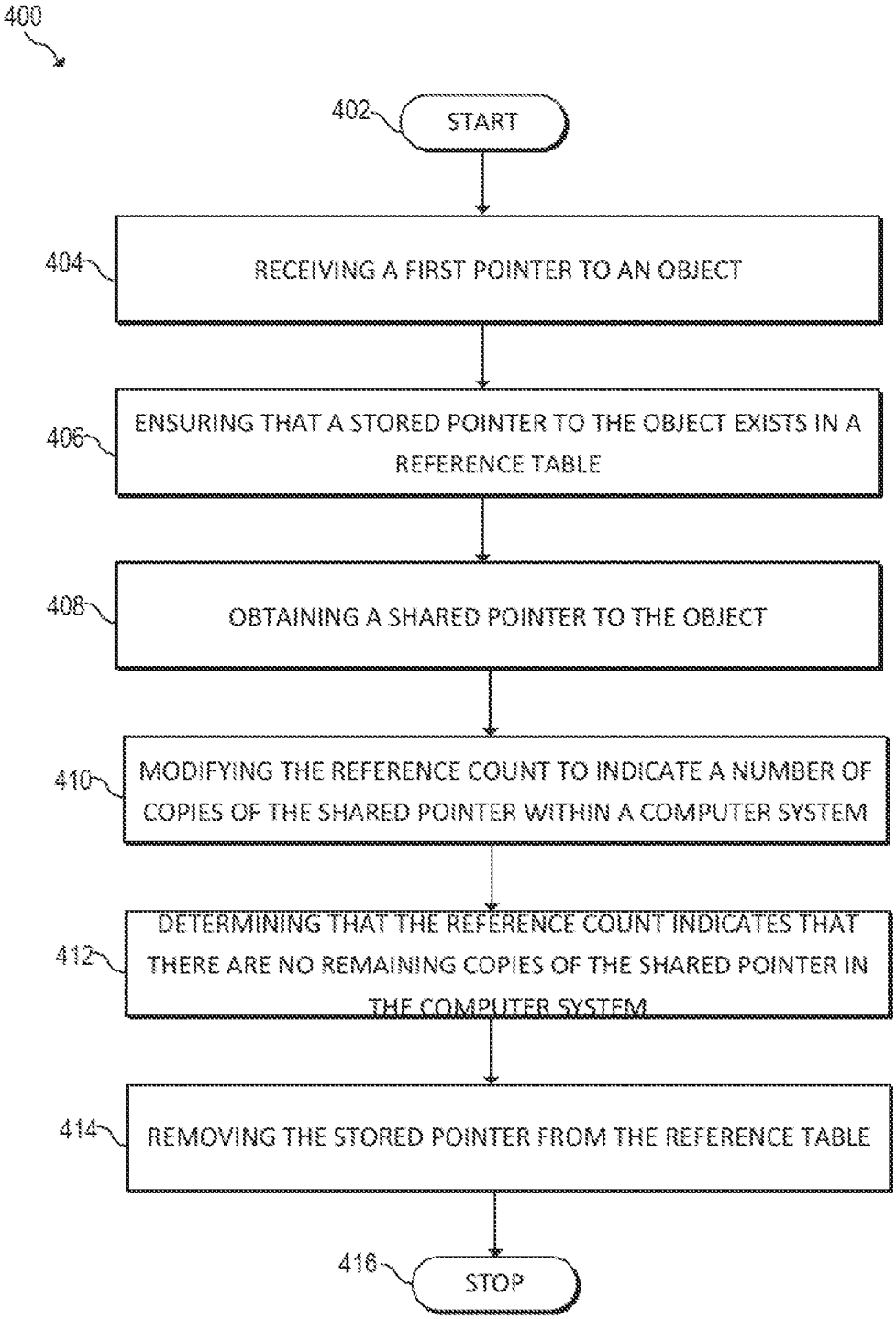
400
402
START
404
RECEIVING A FIRST POINTER TO AN OBJECT
406
ENSURING THAT A STORED POINTER TO THE OBJECT EXISTS IN A REFERENCE TABLE
408
OBTAINING A SHARED POINTER TO THE OBJECT
410
MODIFYING THE REFERENCE COUNT TO INDICATE A NUMBER OF COPIES OF THE SHARED POINTER WITHIN A COMPUTER SYSTEM
412
DETERMINING THAT THE REFERENCE COUNT INDICATES THAT THERE ARE NO REMAINING COPIES OF THE SHARED POINTER IN THE COMPUTER SYSTEM
414
REMOVING THE STORED POINTER FROM THE REFERENCE TABLE
416
STOP

FIG. 4

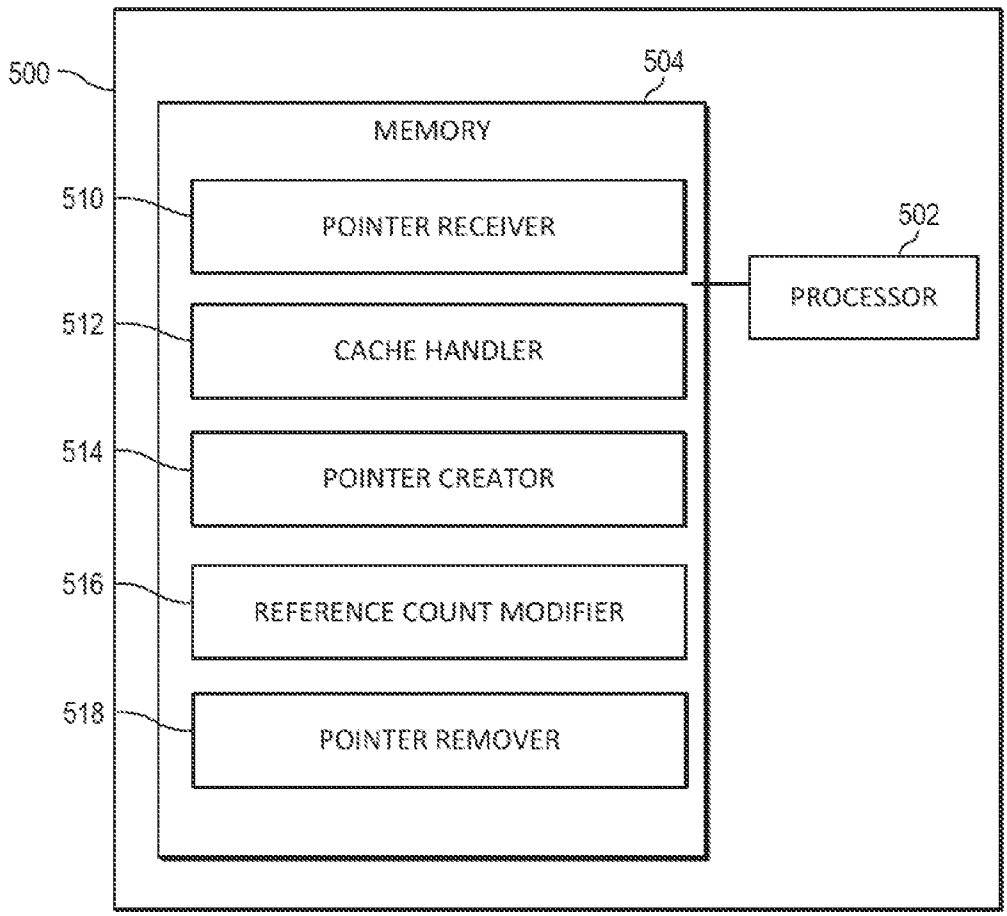
500
504
MEMORY
510
POINTER RECEIVER
502
PROCESSOR
512
CACHE HANDLER
514
POINTER CREATOR
516
REFERENCE COUNT MODIFIER
518
POINTER REMOVER

FIG. 5

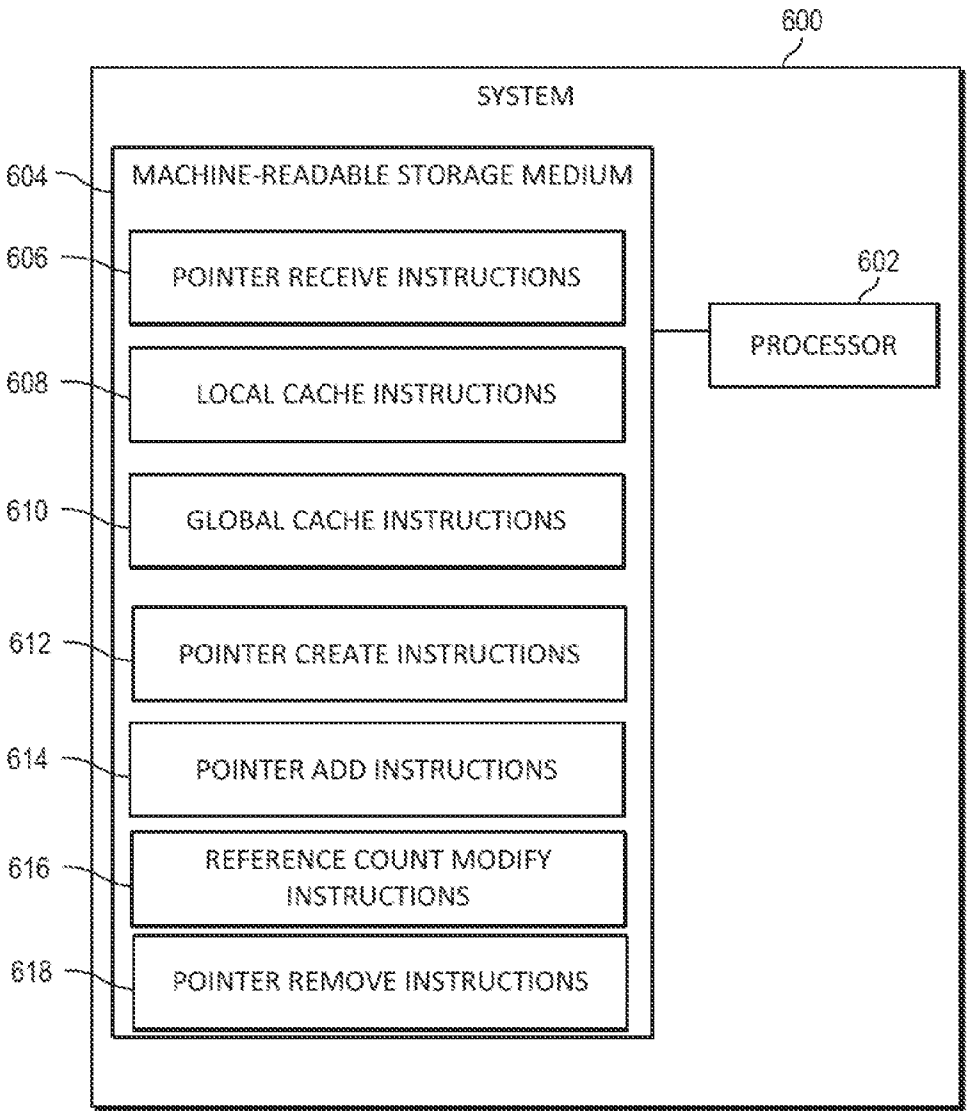
600
SYSTEM
604
MACHINE-READABLE STORAGE MEDIUM
606
POINTER RECEIVE INSTRUCTIONS
608
LOCAL CACHE INSTRUCTIONS
610
GLOBAL CACHE INSTRUCTIONS
612
POINTER CREATE INSTRUCTIONS
614
POINTER ADD INSTRUCTIONS
616
REFERENCE COUNT MODIFY INSTRUCTIONS
618
POINTER REMOVE INSTRUCTIONS
602
PROCESSOR

FIG. 6

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/062931

A. CLASSIFICATION OF SUBJECT MATTER

G06F 12/02(2006.01)i, G06F 12/00(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 12/02; G06F 17/30; G06T 1/60; G06F 12/00; G06F 9/00

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: shared, stored, pointer, object, reference, table, count, garbage, collection, memory

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2014-0071144 A1 (YANG NI et al.) 13 March 2014 See paragraphs [0024], [0026]; and figure 2.	1-3,10-11,13
A		4-9,12,14-15
Y	US 05355483 A (BERTRAND SERLET) 11 October 1994 See column 2, lines 34-42; column 8, lines 55-58; and figure 6.	1-3,10-11,13
A	US 2014-0025651 A1 (IVAN SCHRETER) 23 January 2014 See paragraphs [0039]-[0045]; and figures 4A-4J.	1-15
A	US 04912629 A (ROBERT L. SHULER, JR.) 27 March 1990 See column 6, lines 30-40; and figure 3.	1-15
A	US 2007-0244942 A1 (STEPHEN MCCAMANT et al.) 18 October 2007 See paragraphs [0039]-[0087]; and figure 2.	1-15

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
26 July 2016 (26.07.2016)	**26 July 2016 (26.07.2016)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea
Facsimile No. +82-42-481-8578

Authorized officer
HAN, JOONG SUB
Telephone No. +82-42-481-3578

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2015/062931

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2014-0071144 A1	13/03/2014	CN 102959504 A	06/03/2013
		CN 102959504 B	03/12/2014
		EP 2691852 A1	05/02/2014
		EP 2691852 A4	31/12/2014
		KR 10-1253012 B1	16/04/2013
		KR 10-2012-0123127 A	07/11/2012
		TW 201239633 A	01/10/2012
		TW I471730 B	01/02/2015
		US 2012-254497 A1	04/10/2012
		US 2015-186273 A1	02/07/2015
		US 8566537 B2	22/10/2013
		US 8862831 B2	14/10/2014
		WO 2012-134557 A1	04/10/2012
US 05355483 A	11/10/1994	None	
US 2014-0025651 A1	23/01/2014	None	
US 04912629 A	27/03/1990	None	
US 2007-0244942 A1	18/10/2007	US 7962901 B2	14/06/2011