Title: EXECUTION OF MODIFIED CHENEY SCANNING IN A MULTITHREADED PROCESSING ENVIRONMENT

Abstract: In one embodiment a memory management method uses a third finger fetching pointer thread operating on memory objects distributed in a second memory space between a scan pointer thread and an allocation pointer thread to fetch objects in the second memory space prior to them being scanned. The fetching thread is moved to a next memory slot in the second memory space when the scanned memory slot does not hold a pointer to an object in the first memory space. The memory object is prefetched into a cache shared with an allocation pointer thread when the scanned memory slot holds a pointer to an object in the first space.
EXECUTION OF MODIFIED CHENEY SCANNING IN A
MULTITHREADED PROCESSING ENVIRONMENT

FIELD OF THE INVENTION

[0001] The present invention relates to computing system software. More particularly, this invention relates to memory management.

BACKGROUND

[0002] Reliable operation of software systems generally requires memory management that automatically ensures availability of dynamically allocated memory. Often known as garbage collection, this automatic task is performed by a garbage collector which recycles memory that will never be used again. Automatic garbage collection avoids the need for a programmer to explicitly deallocate memory blocks, and helps eliminate problems associated with memory leaks and premature memory frees. Objects are a collection of fields and methods. The fields can include slots which hold references to other objects, possible recursively. Since garbage collection can also simplify program development by reducing the need to manage storage used to instantiate objects, it is often available in modern languages, including Java and its variants, or C# (Microsoft Common Language Runtime (CLR)).

[0003] However, garbage collection does have a substantial associated runtime overhead, which overhead being in turn dependent on the specific garbage collection algorithm employed. Due to the well known Halting Problem, in general it is not possible for a garbage collector to determine exactly which memory objects are still alive and will be used by the program in the future. All garbage collectors use some approximation to determine aliveness of memory. In a “tracing” garbage collection, objects are determined to be alive if they are reachable. “Copying” garbage collection is a kind of tracing garbage collection that operates by relocating reachable objects (scavenging) and reclaiming objects left behind (such objects being unreachable and therefore dead).

[0004] For example, a two-space copy collector is a simple form of a copying garbage collector that divides available memory into two spaces. Memory objects can be in the first space, also known as the “from” or “old” space. Objects can be moved or allocated in the second space, also known as the “to” or “new” space. The first two spaces is
collectively referred to as the area being collected or C space. Finally an object can be in a third uncollected space or U space. Objects are allocated in in the first space until it is full. This space then becomes the “old” space. The reachable objects are then copied into the other “new” space (usually using a Cheney scan) and the previously allocated memory of the old space is reclaimed for the available memory pool. The Cheney scan is particularly efficient algorithm that requires no stack or other auxiliary space to maintain the list of memory objects not yet scanned. Allocation continues in the new space until it is filled, with the process then being repeated in reverse. As will be appreciated, improvements in copying efficiency of reachable objects translates into improvement in garbage collection and overall system performance.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] The inventions will be understood more fully from the detailed description given below and from the accompanying drawings of embodiments of the inventions which, however, should not be taken to limit the inventions to the specific embodiments described, but are for explanation and understanding only.

[0006] Figure 1 schematically illustrates a computing system supporting multithreaded processing;

[0007] Figure 2 is a Prior Art illustration of operation of a Cheney scan;

[0008] Figure 3 illustrates program logic of a modified Cheney scan using a multiple threads; and

[0009] Figure 4 illustrates operation of a modified Cheney scan using a “third finger” pointer.

DETAILED DESCRIPTION

[0010] Figure 1 generally illustrates a computing system 10 having a processor 12 and memory system 13 (which can be external cache memory, external RAM, and/or memory partially internal to the processor) for executing instructions that can be externally provided in software as a computer program product and stored in data storage unit 18. The processor 12 is capable of supporting multiple threads (a thread is a sequence of
executing instructions that can run independently of other threads yet can directly share data with other threads). Threading support is commonly available in association with computer languages such as Java and C#.

[0011] The computer system 10 of the present invention may include one or more I/O (input/output) devices 15, including a display device such as a monitor. The I/O devices may also include an input device such as a keyboard, and a cursor control such as a mouse, trackball, or trackpad. In addition, the I/O devices may also include a network connector such that computer system 10 is part of a local area network (LAN) or a wide area network (WAN).

[0012] Examples of a system 10 include, but are not limited or restricted to a computer (e.g., desktop, a laptop, a server, blade server, a workstation, a personal digital assistant, etc.) or any peripherals associated therewith; communication equipment (e.g., telephone handset, pager, etc.); a television set-top box and the like. A “connection” or “link” is broadly defined as a logical or physical communication path such as, for instance, electrical wire, optical fiber, cable, bus trace, or even a wireless channel using infrared, radio frequency (RF), or any other wireless signaling mechanism. In addition, the term “information” is defined as one or more bits of data, address, and/or control. “Code” includes software or firmware that, when executed, performs certain functions. Examples of code include an application, operating system, an applet, boot code, or any other series of instructions, or microcode (i.e. code operating at privilege level and below OS).

[0013] Alternatively, the logic to perform the methods and systems as discussed above, could be implemented in additional computer and/or machine readable media, such as discrete hardware components as large-scale integrated circuits (LSI’s), application-specific integrated circuits (ASIC’s), microcode, or firmware such as electrically erasable programmable read-only memory (EEPROM’s); or spatially distant computers relaying information through electrical, optical, acoustical and other forms of propagated signals (e.g., radio waves or infrared optical signals).

[0014] In one embodiment, a computer program product readable by the data storage unit 18 may include a machine or computer-readable medium having stored thereon instructions which may be used to program (i.e. define operation of) a computer (or other electronic devices) to perform a process according to the present invention. The computer-readable medium of data storage unit 18 may include, but is not limited to, floppy
diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAMs), Erasable Programmable Read-Only Memory (EPROMs), Electrically Erasable Programmable Read-Only Memory (EEPROMs), magnetic or optical cards, flash memory, or the like, including any methods to upgrade or reprogram or generate or activate or reserve activation of microcode enhancement)

[0015] Accordingly, the computer-readable medium includes any type of media/machine-readable medium suitable for storing electronic instructions. Moreover, the present invention may also be downloaded as a computer program product. As such, the program may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client). The transfer of the program may be by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem, network connection or the like).

[0016] In one embodiment, the methods of the present invention are embodied in machine-executable instructions directed to control operation of the computing system 10, and more specifically, operation of the processor, register, cache memory, and general memory. The instructions can be used to cause a general-purpose or special-purpose processor that is programmed with the instructions to perform the steps of the present invention. Alternatively, the steps of the present invention might be performed by specific hardware components (including microcode) that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[0017] It is to be understood that various terms and techniques are used by those knowledgeable in the art to describe communications, protocols, applications, implementations, mechanisms, etc. One such technique is the description of an implementation of a technique in terms of an algorithm or mathematical expression. That is, while the technique may be, for example, implemented as executing code on a computer, the expression of that technique may be more aptly and succinctly conveyed and communicated as pseudocode that generically defines program flow logic, by formula, algorithm, or mathematical expression.

[0018] Thus, one skilled in the art would recognize a block denoting \( A + B = C \) as an additive function whose implementation in hardware and/or software would take two
inputs (A and B) and produce a summation output (C). Thus, the use of formula, algorithm, or mathematical expression as descriptions is to be understood as having a physical embodiment in at least hardware and/or software (such as a computer system in which the techniques of the present invention may be practiced as well as implemented as an embodiment).

[0019] Figure 2 is a representation of Cheney scanning as known in the Prior Art. As seen in Figure 2, a classic Cheney scan 20 starts by copying objects that are reachable from the roots into the “to” space. The objects are allocated sequentially using an “allocation pointer” which marks the frontier between the allocated and unallocated areas of the Cheney space. The objects are scanned using a scan pointer that starts with the first object allocated and scans the objects sequentially left to right. If the Cheney Scan encounters a slot referring to an object in the “from” space that already has a “to” space version the slot is updated to refer to the “to” version. If the Cheney Scan encounters an object in the “from” space that has not been moved into the “to” space then a version of the object is allocated it the “to” space using the allocation pointer. The allocation pointer is bumped by the size of the object. Finally the slot holding the “from” object’s address is updated to point to the new “to” version of the object. This technique is sometimes called the two-finger technique where the right finger represents the allocation pointer and the left finger represents the scan pointer. Scanning moves the left finger and allocating moves the right finger. The Cheney scan terminates when the scan pointer catches up with the allocation pointer, in other words the fingers come together. Since “from” space is bounded the algorithm is guaranteed to terminate.

[0020] Unfortunately, in operation the Cheney Scan of Figure 2 frequently encounters an object in the “from” space that is not in the memory cache. The Cheney scan stalls until the object can be pulled into the cache, reducing garbage collection performance.

As seen with reference to procedure 30 of Figure 3, a Cheney scan stall can be eliminated by use of multi-threaded computing systems that share a common cache. A “third finger” is used to assist the scanner by fetching object in the first “from” memory space into the cache ahead of the scan pointer. This third finger fetching thread does its work on objects between the scan pointer and the allocation pointer. It scans the object inspecting each slot. If the slot holds a null pointer or a pointer to an object that is not in an area being collected then it is considered uninteresting and the fetching thread moves to the next slot.
If the slot holds a pointer to an object in “from” space then that object is prefetched into
the cache shared with the allocating thread. If the pointer refers to an object in the second
“to” space then the fetching thread knows that it is “behind” the copying thread and
immediately resumes work with the object being referenced. In other words it skips ahead
to an object that has not been scanned. This technique results in inexpensive non-blocking
coordination between the two threads.

[0021] Figure 4 shows how the fetching thread updates pointers ahead of scan pointer. As
the fetching thread is about to inspect the next object it will notice that if there is a pointer
to an object in the first memory “from” space that has not been copied and it will prefetch
that object so that the scanning thread potentially will not suffer a cache miss.

[0022] On strong memory access ordering machines, the allocating pointer clears the
vtable just past the frontier pointer prior to copying the object so that the fetching thread
will see the null vtable and know that it has reached the frontier pointer. When the
fetching thread encounters a slot holding a reference to an object that has already been
moved then it updates the slot to refer to the “to” version of the object. This helps the
allocating thread by eliminating the need to load the “from” object and then parse the
object header to determine if it has already been moved. On weak memory access ordering
machines the fetching thread must periodically load the allocation pointer and scan up to
the allocation pointer.

[0023] There are several benign race conditions introduced by this procedure. First it is
possible for the allocating thread to move past the fetching thread. The allocating thread is
unaffected. However, the fetching thread will encounter objects with pointers into the “to”
space. This is the only situation where the fetching thread will encounter such pointers. If
the pointer it encounters is to an object to the right of the object being scanned then the
fetching pointer can resume fetching at that object since the allocating thread is at least
that far along.

[0024] As the fetch thread and the allocation thread pass each other it is possible that both
threads can attempt to update a slot in the object being scanned. Since both threads are
attempting to update the pointer with the same value this race condition is benign and the
algorithm remains correct.
[0025] Note that the fetching thread never copies an object or updates the allocation pointer. This job is reserved for the allocating thread so we do not have to synchronize the allocation pointer. The fetching thread can pass the allocation pointer and this will be detected since memory beyond the allocation pointer is zeroed. Since the allocation pointer will never point past initialized memory the worse that can happen is that the fetching thread will get an out of date allocation pointer and stop scanning prematurely. There is nothing that the fetching thread must do to ensure that the algorithm is correct, it merely attempt to off load work that is usually done by the allocating thread, thus speeding up the allocation thread.

[0026] To better understand this procedure, the following pseudocode illustrates one possible implementation scheme for the modified Cheney scan:

```c
// Allocating thread will clear vtable at frontier pointer.
void FetchingThread() {
    while (the fetch pointer := allocation_ptr) {
        Fetch_ptr = Fetch_object_scan(fetch_ptr);
    }
}

void FetchObjectScan(java.lang.Object *obj) {
    java.lang.Object *slot = first_slot(obj);
    java.lang.Object *result = obj;
    // Inspect all slots in the object.
    while (*slot) {
        if (*slot is in a collected area) { // is the object interesting
            if (*slot has already been forwarded) {
                *slot = get_forwarded_obj_for_fetch(slot);
            } else if (*slot has already been updated.) {
                // we are behind the allocating thread
                // skip to the right to try and catchup.
                if (*slot > slot) {
                    // *slot has been allocated and is ahead of us so skip to
                    // scanning that object.
                    return *slot;
                } else {
                    // object is behind so move ahead as quickly as possible.
                    result += get_object_size(obj);
                    return result;
                }
            } else { // *slot is uninteresting since it is not in a collected area.
                slot = get_next_slot (obj);
            }
        }
        result += get_object_size(obj);
        return result;
    }
    java.lang.Object *get_forwarded_obj_for_fetch(java.lang.Object *obj) {
        if (is_forwarded(*slot)) {
            return get_forwarded(*slot);
        } else {
            // fetch the object for the allocating thread.
            unsigned int size = get_object_size(*slot);
            prefetch(*slot, size);
        }
    }
}
```
Java java_lang_Object *get_forwarded_obj_for_alloc_thread
(Java java_lang_Object *obj)
{
    if (is_forwarded(*slot)) {
        return get_forwarded(*slot);
    } else {
        Java java_lang_Object *forwarded_ptr = MARK_FORWARD(allocation_ptr);
        unsigned int size = get_object_size(*slot);
        // we need to copy this object.
        memcpy(*slot, allocation_ptr, size);
        Allocation_ptr += size;
        *allocation_Ptr = NULL; // Clear next field (the vtable) to indicate
        // allocation frontier to prefetching thread.
        *slot = forwarded_ptr;
    }
}

void Allocating_Thread () {
    While (allocation_pointer != scanning_pointer) {
        scanning_ptr = cheney_scan_object(scanning_ptr);
    }
}

// Return a pointer to the next object to scan.
void* CheneyScanObject(Java java_lang_Object *obj) {
    Java java_lang_Object *slot = first_slot(obj)
    Java java_lang_Object *result = obj;
    while (*slot) {
        if (*slot is in collected area) { // is the object interesting
            if (is_forwarded_p(*slot)) {
                *slot = **slot;
            } else if (*slot has not been moved) {
                // Move it. Remember where we moved it and update the slot.
                memcpy (*slot, allocation_ptr, get_object_size (*slot));
                **slot = allocation_ptr; // Remember to version in from
                *slot = allocation_ptr;
                // Update the slot
                allocation_ptr += get_object_size(*slot);
            } else {
                // *slot is not interesting.
            }
        }
    }
    result += get_object_size(obj);
    return result;
}

[0027] Reference in the specification to "an embodiment," "one embodiment," "some
embodiments," or "other embodiments" means that a particular feature, structure, or
characteristic described in connection with the embodiments is included in at least some
embodiments, but not necessarily all embodiments, of the invention. The various
appearances "an embodiment," "one embodiment," or "some embodiments" are not
necessarily all referring to the same embodiments.

[0028] If the specification states a component, feature, structure, or characteristic "may",
"might", or "could" be included, that particular component, feature, structure, or
characteristic is not required to be included. If the specification or claim refers to "a" or
"an" element, that does not mean there is only one of the element. If the specification or claims refer to "an additional" element, that does not preclude there being more than one of the additional element.

[0029] Those skilled in the art having the benefit of this disclosure will appreciate that many other variations from the foregoing description and drawings may be made within the scope of the present invention. Accordingly, it is the following claims including any amendments thereto that define the scope of the invention.
The claimed invention is:

1. A memory management method comprising:
   having a fetching pointer operate on memory objects distributed in a second
   memory space between a scan pointer and an allocation pointer, scanning memory objects
   to find slots holding pointers to objects in the first memory space,
   moving the fetching pointer to a next memory slot in the second memory space
   when the scanned memory slot does not hold a pointer to an object in the first memory
   space, and
   prefetching the memory object in the first space into a cache shared with thread
   supporting the allocation pointer when the scanned memory slot holds a pointer to an
   object in the first space.

2. The memory management method of claim 1, wherein the fetching pointer is
   moved to the next memory slot upon scanning a null pointer.

3. The memory management method of claim 1, wherein the fetching pointer is
   moved to the next memory slot when a pointer to an object that is not in an area being
   collected is encountered.

4. The memory management method of claim 1, further comprising updating the
   fetching pointer to equal the scanning pointer when a slot is encountered that holds a
   pointer to an object in the second space.

5. The memory management method of claim 1, wherein the thread supporting the
   allocating pointer clears a vtable just past the frontier pointer prior to copying the memory
   object to make the fetching thread see a null vtable.

6. A method of garbage collection comprising:
   dividing collected memory into first and second memory spaces,
   scanning the second memory space with a fetching pointer, a scan pointer, and an
   allocation pointer to determine which memory objects to move to the second memory
   space;
   moving the fetching pointer to a next memory slot in the second memory space
   when the scanned memory slot does not hold a pointer to an object in the first memory
   space, and
prefetching the memory object into a cache shared with thread supporting the allocation pointer when the scanned memory slot holds a pointer to an object in the first space.

7. The memory management method of claim 5, wherein the fetching pointer is moved to the next memory slot upon scanning a null pointer.

8. The memory management method of claim 5, wherein the fetching pointer is moved to the next memory slot when a pointer to an object that is not in an area being collected is scanned.

9. The memory management method of claim 5, further comprising referring the pointer to an object in the second memory space when the fetching pointer is behind the copying pointer thread.

10. The memory management method of claim 5, wherein the thread supporting the allocating pointer clears a vtable just past the frontier pointer prior to copying the memory object to make the fetching thread see a null vtable.

11. An article comprising a storage medium having stored thereon instructions that when executed by a machine result in memory management comprising:

   having a fetching pointer operate on memory objects distributed in a second memory space between a scan pointer and an allocation pointer, scanning memory objects to find slots holding pointers to objects in the first memory space,

   moving the fetching pointer to a next memory slot in the second memory space when the scanned memory slot does not hold a pointer to an object in the first memory space, and

   prefetching the memory object in the first space into a cache shared with thread supporting the allocation pointer when the scanned memory slot holds a pointer to an object in the first space.

12. The article comprising a storage medium having stored thereon instructions of claim 11, wherein the fetching pointer is moved to the next memory slot upon scanning a null pointer.
13. The article comprising a storage medium having stored thereon instructions of claim 11, wherein the fetching pointer is moved to the next memory slot when a pointer to an object that is not in an area being collected is encountered.

14. The article comprising a storage medium having stored thereon instructions of claim 11, further comprising updating the fetching pointer to equal the scanning pointer when a slot is encountered that holds a pointer to an object in the second space.

15. The article comprising a storage medium having stored thereon instructions of claim 11, wherein the thread supporting the allocating pointer clears a vtable just past the frontier pointer prior to copying the memory object to make the fetching thread see a null vtable.

16. An article comprising a storage medium having stored thereon instructions that when executed by a machine result in garbage collection comprising:

   dividing collected memory into first and second memory spaces,

   scanning the second memory space with a fetching pointer, a scan pointer, and an allocation pointer to determine which memory objects to move to the second memory space;

   moving the fetching pointer to a next memory slot in the second memory space when the scanned memory slot does not hold a pointer to an object in the first memory space, and

   prefetching the memory object into a cache shared with thread supporting the allocation pointer when the scanned memory slot holds a pointer to an object in the first space.

17. The article comprising a storage medium having stored thereon instructions of claim 16, wherein the fetching pointer is moved to the next memory slot upon scanning a null pointer.

18. The article comprising a storage medium having stored thereon instructions of claim 16, wherein the fetching pointer is moved to the next memory slot when a pointer to an object that is not in an area being collected is scanned.
19. The article comprising a storage medium having stored thereon instructions of claim 16, further comprising referring the pointer to an object in the second memory space when the fetching pointer is behind the copying pointer thread.

20. The article comprising a storage medium having stored thereon instructions of claim 16, wherein the thread supporting the allocating pointer clears a `vtable` just past the frontier pointer prior to copying the memory object to make the fetching thread see a null `vtable`.

21. A memory management system comprising:

   a processor having a connected memory, with the processor executable to operate on memory objects in the connected memory, the memory objects being distributed in a second memory space between a scan pointer and an allocation pointer, the system operating to scan memory objects to find slots holding pointers to objects in a first memory space,

   move a fetching pointer to a next memory slot in the second memory space when the scanned memory slot does not hold a pointer to an object in the first memory space, and

   prefetch the memory object in the first space into a cache shared with thread supporting the allocation pointer when the scanned memory slot holds a pointer to an object in the first space.

22. The memory management method of claim 21, wherein the fetching pointer is moved to the next memory slot upon scanning a null pointer.

23. The memory management system of claim 21, wherein the fetching pointer is moved to the next memory slot when a pointer to an object that is not in an area being collected is encountered.

24. The memory management system of claim 21, further comprising updating the fetching pointer to equal the scanning pointer when a slot is encountered that holds a pointer to an object in the second space.

25. The memory management method of system 21, wherein the thread supporting the allocating pointer clears a `vtable` just past the frontier pointer prior to copying the memory object to make the fetching thread see a null `vtable`.
26. A garbage collection system comprising:

   a processor having a connected memory, with the processor executable to operate on memory objects in the connected memory, and divide collected memory into first and second memory spaces,

   scan the second memory space with a fetching pointer, a scan pointer, and an allocation pointer to determine which memory objects to move to the second memory space;

   move the fetching pointer to a next memory slot in the second memory space when the scanned memory slot does not hold a pointer to an object in the first memory space,

   and

   prefetch the memory object into a cache shared with thread supporting the allocation pointer when the scanned memory slot holds a pointer to an object in the first space.

27. The memory management method of claim 26, wherein the fetching pointer is moved to the next memory slot upon scanning a null pointer.

28. The memory management method of claim 26, wherein the fetching pointer is moved to the next memory slot when a pointer to an object that is not in an area being collected is scanned.

29. The memory management method of claim 26, further comprising referring the pointer to an object in the second memory space when the fetching pointer is behind the copying pointer thread.

30. The memory management method of claim 26, wherein the thread supporting the allocating pointer clears a vtable just past the frontier pointer prior to copying the memory object to make the fetching thread see a null vtable.
FROM Space - Linked Data Structures - Hard to prefetch

TO Space - Copying smashes cache - we are bus limited

Scanned

Allocated but unscanned

Free

Not needed anymore

Larger than the cache can hold

No fill needed

Scan Pointer

Allocation Pointer

Fig. 2
(Prior Art)
Third finger fetching thread operates on objects between the scan pointer and the allocation pointer, scanning the object while inspecting each slot.

Does slot hold a null pointer or a pointer to an object that is not in an area being collected?

Yes

Move fetching thread to the next slot

No

If the slot holds a pointer to an object in "from" space, prefetch object into the cache shared with the allocating thread.

If the pointer refers to an object in the "To" space then the fetching thread knows that it is "behind" the copying thread and immediately resumes work with the object being referenced.

Fig. 3
FROM Space - Linked Data Structures - Hard to prefetch

TO Space - Fetch pointer updates ptrs ahead of scan

Scanned  Allocated but unscanned  Free

Not needed anymore  Larger than the cache can hold  No fill needed

Scan Pointer  Fetch Pointer  Allocation Pointer

Fig. 4