LOAD BALANCING

BACKGROUND

[0001] Memory management in a managed execution environment may include temporarily suspending managed threads during a sweep or scan of a memory heap. However, a prolonged suspension of the managed threads may time-out execution of the corresponding application.

SUMMARY

[0002] Execution of portions of an application, program, method, function, or other assemblage of code in a managed execution environment may be influenced by, at least, the potential magnitude of a memory management implementation.

DESCRIPTION OF THE DRAWINGS

[0003] Load balancing in a managed execution environment is presently described in accordance with the following figures.

[0004] FIG. 1 shows devices communicating over a network, with the devices implementing example technologies for load balancing.

[0005] FIG. 2 shows an example of an execution environment for implementing example technologies for load balancing.

[0006] FIG. 3 shows an example data flow among data nodes in accordance with an example implementation of load balancing.

[0007] FIG. 4 shows an example processing flow in accordance with an example implementation of load balancing.

DETAILED DESCRIPTION

[0008] Load balancing is described herein. More particularly, the present description relates to controlling certain behavior of at least portions of an application, program, method, function, or other assemblage of code based on data regarding, at least, the potential magnitude of a memory management implementation in the respective execution environment.

[0009] Within the context of this detailed description and as pertaining to object-oriented programming, a method may be regarded as the processing that an object performs. Therefore, for instance, when a message is sent to an object, the method is implemented.

[0010] "Process," as described herein, may refer to the actual running of a module or assembly of code pertaining to an application, program, function, or other assemblage of programmable and executable code.

[0011] "Assembly," as described herein, may refer to a unit of deployment of code, which may or may not be versionable.

[0012] "Resource," as described herein, may include both physical and logical resources associated with a given computing environment. As non-limiting examples, such resources may range from files to ports to shared state; that is, any non-executable entity that may be shared by more than one executable entity.

[0013] "Threads," as described herein, may refer to execution paths within an application, program, function, or other assemblage of programmable and executable code. Threads enable multiple paths or streams of execution of modules of executable instructions to occur concurrently within the same application, program, function, or other assemblages of programmable and executable code; whereby, within each stream, a different transaction or message may be processed. A multitasking or multiprocessing environment, in which multi-threading processes may be executed, may be found in either a managed execution environment or an unmanaged execution environment.

[0014] "Isolation boundary," as described herein, may refer to a logical or physical construct that may serve as a unit of isolation. Processes are an example of an isolation boundary. Within a managed execution environment, such an isolation boundary may be referred to as an application domain, in which multiple threads of execution may be contained. Such terminology is provided as an example only. That is, the example implementations described herein are not limited to application domains or even to managed execution environments as stated above, but rather may be applied within various other isolation boundaries implementations in other execution environments. More particularly, isolation boundaries, as related to the scope of resource distribution described herein, may further pertain to machine boundaries, process boundaries, threads, and class or assembly boundaries. Even more particularly, the scope of resource distribution may pertain to public/private exposure, assemblies, or classes. Further, resource distribution may have multiple axis or annotations including, *e.g.*, a type of resource as well as visibility of the resource.

[0015] Isolation boundaries may enable the code to be executed therein to be loaded from a specified source; an isolation boundary may be aborted independent of other such isolation boundaries; and processing within an isolation boundary may be isolated so that a fault occurring therein does not affect other isolation boundaries within the process. More particularly, isolation boundaries may isolate the consumption of resources therein to the extent that other isolation boundaries either do not see any changes to a resource or, rather, see the resources in a serialized, atomic fashion.

[0016] FIG. 1 shows example network environment 100 having processing nodes that may communicate with each other in various quantities and combinations thereof to implement one or more examples of load balancing. However, implementations of load balancing are not limited to nodes in a network environment. Still, in the example network environment of FIG. 1, client device 105, server devices

110A and 110B, and "other" device 115 may be communicatively coupled to one another via network 125; and, further, at least one of client device 105, server devices 110A and 110B, and "other" device 115 may be capable of implementing load balancing 120, as described herein.

[0017] Client device 105 may be at least one of a variety of conventional computing devices, including a desktop personal computer (PC), workstation, mainframe computer, Internet appliance, and set-top box. Further, client device 105 may be at least one of any device that is capable of being associated with network 125 by a wired and/or wireless link, including a personal digital assistant (PDA), laptop computer, cellular telephone, *etc.* Further still, client device 105 may represent the client devices described above in various quantities and/or combinations thereof. "Other" device 115 may also be embodied by any of the above examples of client device 105.

[0018] Server devices 110A and 110B may provide any of a variety of data and/or functionality to client device 105 or "other" device 115 in accordance with at least one implementation of load balancing 120. The data may be publicly available or alternatively restricted, *e.g.*, restricted to only certain users or only if an appropriate subscription or licensing fee is paid. Either of server devices 110A and 110B may be a network server, an application server, or a blade server, in various quantities and combinations thereof. Typically, server devices 110A and 110B serve as content sources, and client device 105 receives such content either via network 125 or in an off-line manner. However, according to the example implementations described herein, client device 105 and server devices 110A and 110B may interchangeably be sending nodes or receiving nodes in network environment 100. Further, according to at least one example of load balancing 120, server devices 110A and 110B may be implemented as one of many server devices on network 125. Such configuration may be regarded, informally, as a "server farm." In such a server farm, the two or more networked servers may share in

the execution of at least portions of a same application, program, function, or other assemblage of programmable and executable code. Further still, "other" device 115 may also be embodied by any of the above examples of server devices 110A and 110B.

[0019] "Other" device 115 may be any further device that is capable of implementing load balancing 120 according to one or more of the examples described herein. That is, "other" device 115 may be a software-enabled computing or processing device that is capable of implementing load balancing 120 for at least a portion of an application, program, function, or other assemblage of programmable and executable code in at least a managed execution environment. More particularly, "other" device 115 may serve as a load balancer. As a load balancer, "other" device 115 may be implemented as a hardware device that is physically separate from either of client device 105 or server devices 110A and 110B; alternatively, "other" device 115 may be implemented as firmware or software on at least one of client device 105 or server devices 110A and 110B. Thus, "other" device 115 may be a computing, processing, or server device having at least one of an operating system, an interpreter, converter, compiler, or runtime execution environment implemented thereon. These examples are not intended to be limiting in any way, and therefore should not be construed in that manner.

[0020] Network 125 may represent any of a variety of conventional network topologies and types, which may include wired and/or wireless networks. Network 125 may further utilize any of a variety of conventional network protocols, including public and/or proprietary protocols. Network 125 may include, for example, the Internet as well at least portions of one or more local area networks (also referred to, individually, as a "LAN"), such as an 802.11 system; a personal area network (*i.e.*, PAN), such as Bluetooth.

[0021] Computer architecture in at least one of devices 105, 110A, 110B, and 115 has typically defined computing platforms in terms of hardware and software.

Software for computing devices came to be categorized into groups, based on function, which include: a hardware abstraction layer (alternatively referred to as a "HAL"), an operating system (alternatively referred to as "OS"), and applications.

[0022] A runtime execution environment may refer to an isolated space, which may be between the OS and an application, in which the application may execute specific tasks on at least one of processing device 105, one or more of server devices 110A and 110B, or other device 115. More particularly, a runtime execution environment is intended to enhance the reliability of the execution of applications on a growing range of processing devices including servers, desktop computers, laptop computers, mobile processing devices, set-top boxes, and gaming consoles by providing a layer of abstraction and services for an application running on such processing devices, and further providing the application with capabilities including memory management and configuration thereof.

[0023] A runtime execution environment may serve as at least one of an application programming and application execution platform.

[0024] As an application programming platform, a runtime execution environment may compile targeted applications, which may be written in one of multiple computing languages, into an intermediate language (hereafter "IL"). IL is typically independent of the platform and the central processing unit (hereafter "CPU") executes IL. In fact, IL is a higher level language than many CPU machine languages.

[0025] As an application execution platform, a runtime execution environment may interpret compiled IL into native machine instructions. A runtime execution environment may utilize either an interpreter or a "just-in-time" (hereafter "JIT") compiler to execute such instructions. Alternatively, a runtime execution environment may execute, read, interpret, or otherwise analyze intermediate language code (*i.e.*, "IL") that is distributed to the runtime execution environment in IL format rather than in a native

platform execution format and is already compiled into any one of assemblies, methods, or types. A source of such IL may be disposed in either of a non-managed execution environment or a separate implementation of the runtime execution environment on a same or separate one of devices 105, 110, and 115. The source may deploy the IL at, or prior to, install time for the application, program, method, function, or other assemblage of programmable and executable code to which the IL corresponds.

[0026] Regardless, the native machine instructions may then be directly executed by the CPU. Since IL is CPU-independent, IL may execute on a CPU platform as long as the OS running on that CPU platform hosts an appropriate runtime execution environment. Examples of runtime environments, to which load balancing 120 may pertain, include: Visual Basic runtime environment; Java® Virtual Machine runtime environment that is used to run, *e.g.*, Java® routines; or Common Language Runtime (CLR) to compile, *e.g.*, Microsoft .NET™ applications into machine language before executing a calling routine. However, such listing provides examples only. The example implementations are not limited to just these managed execution environments. Further, the example implementations are not just limited to managed execution environments, for one or more examples may be implemented within testing environments and/or unmanaged execution environments.

[0027] An application compiled into IL may be referred to as "managed code," and therefore a runtime execution environment may be alternatively referred to as a "managed execution environment." Portions of managed code may be referred to as a "managed image." Code that does not utilize a runtime execution environment to execute may be referred to as native code applications.

[0028] FIG. 2 shows an example of runtime execution environment 200 in which examples of load balancing 120 (see FIG. 1) may be implemented.

[0029] According to at least one example implementation, runtime execution environment 200 may facilitate execution of managed code for a computing device platform. Managed code may be considered to be part of a core set of application-development technologies, and may further be regarded as code that is compiled for execution on runtime execution environment 200 to provide a corresponding service to the computing device platform. In addition, runtime execution environment 200 may translate managed code at an interpretive level into instructions that may be proxied and then executed by a processor. A framework for runtime execution environment 200 may also provide class libraries, which may be regarded as software building blocks for managed applications.

[0030] According to a further example implementation, runtime execution environment 200 may provide at least partial functionality that may otherwise be expected from a kernel, which may or may not be lacking from a computing device platform depending upon resource constraints for any particular one of devices 105, 110A, 110B, and 115. Thus, at least one example of runtime execution environment 200 may implement the following: input/output (hereafter "I/O") routine management, memory management, administration, and service routine management. Thus, runtime execution environment 200 may include I/O component 205, at least one memory manager 210, administrator 215, and execution component 220. These components, which are to be described in further detail below, are provided as examples only; that is, the examples are not intended to be limiting to any particular implementation, and no such inference should be made. Further, the components may be implemented in examples of runtime execution environment 200 in various combinations and configurations thereof.

[0031] I/O component 205 of runtime execution environment 200 may provide asynchronous access to data sources (*i.e.*, processor and peripherals) associated with the

computing device platform. Examples of such data sources may be one or more of devices 105, 110A, 110B, and 115, described above with regard to FIG. 1. More particularly, I/O component 205 may provide runtime execution environment 200 with robust system throughput and further streamline performance of code from which an I/O request originates.

[0032] Memory manager 210 may refer to a module within or associated with runtime execution environment 200 that is regarded as a "garbage collector." Garbage collection (alternatively referred to hereafter as "GC") may be regarded as a robust feature of managed code execution environments by which an object is freed (*i.e.*, de-allocated) if an object is no longer used by any applications, upon a sweep or scan of a memory heap. In at least one example of memory manager 210, a sweep of free memory heap may be implemented as a linear search. Such implementation may be well-suited for an example of a computing device platform for which memory size is constrained and for which a delay in completion of a sweep may be perceived by a user of a corresponding device.

[0033] An example of memory manager 210 may implement "concurrent GC" functionality to allow managed threads to continue to run during a sweep or scan of a memory heap. That is, during de-allocation, concurrent GC may allow allocation to continue in parallel. Concurrent GC functionality (alternatively referred to herein as "concurrent GC") may be suitable for an interactive or GUI (graphical user interface)-based application.

[0034] However, although load balancing 120 may be relevant to concurrent GC functionality, the present example implementations of load balancing 120 are described in the context of non-concurrent GC functionality, by which one or more managed threads may be suspended during a sweep or scan of a memory heap.

[0035] Further functions implemented by memory manager 210 may include: managing one or more contiguous blocks of finite volatile RAM (*i.e.*, memory heap) storage or a set of contiguous blocks of memory amongst the tasks running on the computing device platform; allocating memory to at least one application running on the computing device platform; freeing at least portions of memory on request by at least one of the applications; and preventing any of the applications from intrusively accessing memory space that has been allocated to any of the other applications.

[0036] Administrator 215 may refer to a module within or associated with runtime execution environment 220 that serves to receive at least a portion of an application, program, method, function, or other assemblage of programmable and executable code for execution in runtime execution environment 200. Further, in accordance with at least one example implementation of load balancing 120, administrator 215 may coordinate with memory manager 210 to control the behavior of the application, program, method, function, or other assemblage of programmable and executable code within runtime execution environment 220, at compile time, initial runtime, or at any time thereafter during execution of an application.

[0037] Execution component 220 may refer to a module within or associated with runtime execution environment 200 that serves to enable execution of managed code for the computing device platform. Execution component 220 may be regarded as the environment in which execution of the code of the application is implemented, and in which runtime services (*e.g.*, device access and memory management) may be provided.

[0038] FIG. 3 shows example data flow 300 in accordance with an example implementation of load balancing 120 (see FIG. 1). Example data flow 300 is presently described with references to features from FIGS. 1 and 2, although such implementations are provided only as examples and are not intended to be construed in any limiting manner.

[0039] Processes 305A, 305B, and 305C may be regarded as logical or physical constructs, in accordance with at least the description of "isolation boundaries" provided above. Thus, processes 305A, 305B, and 305C may pertain to machine boundaries, process boundaries, threads, application domains, and class or assembly boundaries, in accordance with various implementations of load balancing 120. Further, the implementations of load balancing 120 are in no way limited to three such constructs (*i.e.,* 305A, 305B, and 305C), which are illustrated and explained herein only for descriptive purposes.

[0040] As an example, processes 305A, 305B, and 305C may pertain to multiple servers (*i.e.,* machine boundaries). Implemented as hardware, servers 305A, 305B, and 305C may be separate hardware devices, on which at least one runtime execution environment 200 is running as an application execution platform. Implemented as software servers, processes 305A, 305B, and 305C may be modularized components of a common application, program, method, function, or other assemblage of programmable and executable code being executed by an implementation of runtime execution environment 200.

[0041] Memory management data 310, originating from at least one of processes 305A, 305B, and 305C, may include data referring to an impending sweep or scan of a memory heap (*i.e.,* GC) to free (*i.e.,* de-allocate) one or more objects that are no longer used by an application, program, method, function, or other assemblage of code being executed by an implementation of runtime execution environment 200.

[0042] By the example in which processes 305A, 305B, and 305C are implemented as separate hardware devices, memory management data 310 may originate from an implementation of runtime execution environment 200 running on one or more of such devices. By the example in which processes 305A, 305B, and 305C are implemented as modularized components of a common application, program, method,

function, or other assemblage of programmable and executable code, me mory management data 310 may originate from an implementation of runtime execution environment 200 on which the modularized components are being executed. More particularly, by the above examples, memory management data 310 may originate from memory manager 210 or administrator module 215, either singularly or combined.

[0043] Memory management data 310 may implement transactions by using one or more application program interfaces (API) that are compatible with APIs of various system architectures. More particularly, the APIs corresponding to memory management data 310 may be capable of initiating load balancing among processes 305A, 305B, and 305C (*i.e.,* across machine boundaries, process boundaries, threads, application domains, and class or assembly boundaries) in accordance with various implementations of load balancing 120. As described herein, an API may be regarded as one or more routines used by an application, program, method, function, or other assemblage of programmable and executable code to direct the performance of procedures by runtime execution environment 200 or even an operating system.

[0044] Memory management data 310 may include one or more APIs implemented by memory manager 210 or administrator module 215, either singularly or combined. The nomenclature for such APIs listed and described below is provided for descriptive purposes, and are provided as non-limiting examples.

[0045] MaxGenerationNotifier may refer to a class of which instantiations may be intended to prompt load balancer 315 to redistribute at least a portion of the same or another application, program, method, function, or other assemblage of code being executed by a particular implementation of runtime execution environment 200. More particularly, MaxGenerationNotifier may be instantiated when memory manager 210 or administrator 215, either singularly or combined, detect that an impending GC on runtime execution environment 200 may run so long as to cause a time-out in the

execution of the same or another application, program, method, function, or other assemblage of code on runtime execution environment 200. Such detection may be based upon the allocation of the threshold amount of memory heap during the current execution of the application, program, method, function, or other assemblage of code. Effectively, then, an instantiation of MaxGenerationNotifier may provide load balancer with a proactive notification of the state of runtime execution environment 200.

[0046] More particularly, MaxGenerationNotifier may be instantiated when memory manager 210 or administrator 215, either singularly or combined, detect that an impending GC may potentially exceed a threshold logical or physical magnitude. For example, MaxGenerationNotifier may be instantiated when memory manager 210 or administrator 215, either singularly or combined, detect that the impending GC is an implementation of a late generation GC, and therefore is likely to sweep or scan the largest amount of heap (*e.g.,* 2 GB) permitted under currently-available processing capabilities. As a result, execution of an application, program, method, function, or other assemblage of code may time-out before completion of the late-generation GC since managed threads may be suspended during a sweep or scan of a memory heap. Thus, before a GC of such magnitude begins, MaxGenerationNotifier may be instantiated to notify load balancer 315 of at least a need to redistribute at least a portion of the execution of the application, program, method, function, or other assemblage of code.

[0047] Of course, the 2GB heap is described above only as an example of a late-generation GC. The magnitude of an impending GC that may act as a catalyst for instantiation of MaxGenerationNotifier may vary based on myriad of factors, and may increase as processing capabilities evolve. Such factors may relate to processing capabilities of the processing device as well as time anticipated for the GC to complete. Further, the preference of the programmer who wrote the executing application, program, method, function, or other assemblage of code, as indicated to administrator

215, may also initiate an instantiation of MaxGenerationNotifier, when the programmer is motivated by at least one of efficiency and security concerns.

[0048] Further still, at least one alternative implementation of MaxGenerationNotifier may include one or more specific instructions for redirecting execution for at least a portion of the corresponding application, program, method, function, or other assemblage of code being executed by runtime execution environment 200.

[0049] MaxGenerationCompleteNotifier may refer to a class of which instantiations are intended to notify load balancer 315 that the GC instantiated by MaxGenerationNotifier has been completed. Therefore, an instantiation of MaxGenerationCompleteNotifier may serve to notify load balancer 315 that redistribution of one or more portions of the executing application, program, method, function, or other assemblage of code may cease without adversely affecting execution by runtime execution environment 200. More particularly, the MaxGenerationNotifier may be instantiated when memory manager 210 or administrator 215, either singularly or combined, detect at least one of: completion of the GC that served as catalyst of a most recent instantiation of MaxGenerationNotifier; a predetermined threshold of time has passed since the instantiation of MaxGenerationNotifier; or another programmed module of code has been executed for the currently executing application, program, method, function, or other assemblage of code.

[0050] Load Balancer 315 may refer to a either a hardware device that is physically separate from either of client device 105 or as firmware or software component on at least one of client device 105 or server devices 110A and 110B.

[0051] Upon an instantiation of MaxGenerationNotifier, load balancer 315 may redistribute processing of at least portions of a currently executing application, program, method, function, or other assemblage of code to avoid a time-out during GC. For

example, upon receiving memory management data 310 which includes an instantiation of MaxGenerationNotifier referring to an impending GC on process 305A, load balancer 315 may receive one or more requests for execution of portions of the same or another currently executing application, program, method, function, or other assemblage of code, and re-direct such execution to at least one of process 305B and process 305C.

[0052] Upon an instantiation of MaxGenerationCompleteNotifier, load balancer 315 may cease redistribution of processing of portions of a currently executing application, program, method, function, or other assemblage of code to process 305B and process 305C. Thus, load balancer 315 may have the option of having the balance of the currently executing application, program, method, function, or other assemblage of code executed on process 305A.

[0053] At least one alternative implementation of data flow 300 does not include load balancer 315. That is, such alternative implementations achieves load balancing by having processes 305A, 305B, and 305C

[0054] FIG. 4 shows example processing flow 400 corresponding to at least one example implementation of load balancing 120 (see FIG. 1). Example processing flow 400 is described below with references to features from FIGS. 1-3, although such implementations are provided only as examples and are not intended to be construed in any limiting manner.

[0055] Block 405 may refer to execution of at least a portion of an application, program, method, function, or other assemblage of code being implemented as part of process 305A, process 305B, or process 305C, either singularly or collaboratively in various combinations. Although load balancing 120 is not so limited, block 405 may refer to execution in a managed execution environment (*i.e.,* runtime execution environment 200).

[0056] Block 410 may refer to memory manager 210 or administrator 215, either singularly or in combination, determining that a threshold amount of heap has already been allocated during execution of a currently executing application, program, method, function, or other assemblage of code, and therefore a GC may be impending. Alternatively, block 410 may refer to memory manager 210 or administrator 215, either singularly or in combination, inspecting at least a portion of the programmed code to determine that a sweep or scan of a memory heap of the processing device upon which runtime execution environment 200 is running may be impending.

[0057] Decision 415 may refer to memory manager 210 or administrator 215, either singularly or in combination, assessing the magnitude of the anticipated GC implementation relative to a predetermined threshold value. The assessment may be made to determine whether the physical magnitude of the memory heap exceeds a predetermined threshold value (*e.g.,* 2GB or more) that would likely time-out execution of a currently executing application, program, method, function, or other assemblage of code. According to one alternative, the assessment may be made of the time anticipated for the GC to complete against a predetermined time threshold for avoiding a time-out. According to yet another example, the assessment may even include inspecting at least a portion of the programmed code to determine whether the impending GC is deliberately intended to serve as a catalyst for at least portions of the executing application, program, method, function, or other assemblage of code to be redistributed.

[0058] Block 420A, subsequent to negative decision 415, may refer to memory manager 210 implementing GC for the executing application, program, method, function, or other assemblage of code on the host one of processes 305A, 305B, or 305C. Processing flow may then return to block 405 to continue execution of the currently executed application, program, method, function, or other assemblage of code.

[0059] Block 425, subsequent to positive decision 415, may refer to memory manager 210 or administrator 215, either singularly or in combination, notifying load balancer 315 that the anticipated GC meets or exceeds the predetermined threshold values. That is, upon determining that a logical or physical parameter of the anticipated GC is expected to exceed a threshold value, management module 210 or administrator 215, either singularly or in combination, may notify load balancer that the impending GC is likely to cause a time-out for a currently executed application, program, method, function, or other assemblage of code. Even more particularly, management module 210 or administrator 215, either singularly or in combination, may instantiate an API (*e.g.,* MaxGenerationNotifier), which provides a notification of anticipated state.

[0060] In at least one alternative implementation of block 425, memory manager 210 or administrator 215, either singularly or in combination, may provide a notification in another form. For example, the notification may be an event that includes calling the application, program, method, function, or other assemblage of code that is executed as part of process 305A, process 305B, or process 305C previously registered with memory manager 210 or administrator 215.

[0061] An example of such a notification event may include each of processes 305A, 305B, and 305C in FIG. 3 subscribing to the one or more memory management events (*e.g.,* MaxGenerationNotifier) via memory manager 210 or administrator 215 either singularly or combined. For example, process 305A may implement two functions that may be called when MaxGenNotifier is issued by memory manager 210 or administrator 215 either singularly or in combination. Non-limiting examples of such calls are as follows:

(1) GC.MaxGenerationNotifier += new MaxGenerationNotificationHandler(On_MaxGeneration_Notify1);

(2) GC.MaxGenerationNotifier += new MaxGenerationNotificationHandler(On_MaxGeneration_Notify2);

whereby MaxGeneration_Notify1 and MaxGeneration_Notify2 are the aforementioned functions. Further, functions MaxGeneration_Notify1 and MaxGeneration_Notify2 may notify load balancer 315 of the subscribed events. Therefore, upon occurrence of a subscribed event, load balancer 315 may take requested load balancing action.

[0062] Block 430 may refer to load balancer 315 redistributing at least portions of the currently executed application, program, method, function, or other assemblage of code among processes 305A, 305B, and 305C. Thus, for example, if the notification at block 425 is received in anticipation of a GC implementation in accordance with host process 305A, block 430 may refer to load balancer 315 receiving a request for execution of at least a portion of the same or another currently executed application, program, method, function, or other assemblage of code, and re-direct the requested execution to at least one of processes 305B and 305C. That is, block 430 may refer to load balancer 315 redistributing at least portions of the currently executed application, program, method, function, or other assemblage of code from currently running process 305A to a concurrently running one of processes 305B and 305C.

[0063] Block 420B may refer to memory manager 210 implementing GC for the executing application, program, method, function, or other assemblage of code on the host one of processes 305A, 305B, or 305C, while at least portions of the respective application, program, method, function, or other assemblage of code are being executed on others of concurrently running processes 305A, 305B, and 305C. Thus, for example, if the notification at block 425 is received in anticipation of a GC implementation in accordance with host process 305A, block 430 may refer to load balancer 315 redistributing at least portions of the currently executed application, program, method,

function, or other assemblage of code to at least one of processes 305B and 305C, and block 420A may refer to memory manager 210 implementing GC on host process 305A.

[0064] Block 435 may refer to memory manager 210 or administrator 215, either singularly or in combination, notifying load balancer 315 that the GC has been completed. The notification may or may not include instructions to cease a current load balancing implementation. Even more particularly, management module 210 or administrator 215, either singularly or in combination, may instantiate an API (*e.g.,* MaxGenerationNotifier), which provides a notification of current state. Processing flow may then return to block 405 to continue execution of a currently executed application, program, method, function, or other assemblage of code.

[0065] Similar to processing at block 425, a notification event may include each of processes 305A, 305B, and 305C in FIG. 3 subscribing to the one or more memory management events (*e.g.,* MaxGenerationCompleteNotifier) via memory manager 210 or administrator 215 either singularly or combined. Thus, for example, process 305A may implement two functions that may be called when MaxGenCompleteNotifier is issued by memory manager 210 or administrator 215 either singularly or in combination. Non-limiting examples of such calls are as follows:

(1) GC.MaxGenerationCompleteNotifier += new MaxGenerationCompleteNotificationHandler(On_MaxGenerationComplete_Notify1);

(2) GC.MaxGenerationCompleteNotifier += new MaxGenerationCompleteNotificationHandler(On_MaxGenerationComplete_Notify2);

whereby MaxGenerationComplete_Notify1 and MaxGenerationComplete_Notify2 are the aforementioned functions. Further, functions MaxGenerationComplete_Notify1 and MaxGenerationComplete_Notify2 may notify load balancer 315 of the subscribed events. Therefore, upon occurrence of a subscribed event, load balancer 315 may at least temporarily cease load balancing, as requested.

[0066] The description above, pertaining to FIGS. 1-4, load balancing with regard to one or more applications executing on a runtime execution environment may be implemented in accordance with data associated with memory management data. However, the example limitations described herein are not limited to just load balancing based on memory management data. Rather, a notification (*i.e.,* notification) may be instantiated to implement load balancing based on myriad logical and physical criteria.

[0067] The computer environment for any of the examples and implementations described above may include a computing device having, for example, one or more processors or processing units, a system memory, and a system bus to couple various system components.

[0068] The computing device may include a variety of computer readable media, including both volatile and non-volatile media, removable and non-removable media. The system memory may include computer readable media in the form of volatile memory, such as random access memory (RAM); and/or non-volatile memory, such as read only memory (ROM) or flash RAM. It is appreciated that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes or other magnetic storage devices, flash memory cards, CD-ROM, digital versatile disks (DVD) or other optical storage, random access memories (RAM), read only memories (ROM), electric erasable programmable read-only memory (EEPROM), and the like, can also be utilized to implement the example computing system and environment.

[0069] Reference has been made throughout this specification to "an example," "alternative examples," "at least one example," "an implementation," or "an example implementation" meaning that a particular described feature, structure, or characteristic is included in at least one implementation of the present invention. Thus, usage of such phrases may refer to more than just one implementation. Furthermore, the described

features, structures, or characteristics may be combined in any suitable manner in one or more implementations.

[0070] One skilled in the relevant art may recognize, however, that the invention may be practiced without one or more of the specific details, or with other methods, resources, materials, etc. In other instances, well known structures, resources, or operations have not been shown or described in detail merely to avoid obscuring aspects of the invention.

[0071] While example implementations and applications of the present invention have been illustrated and described, it is to be understood that the invention is not limited to the precise configuration and resources described above. Various modifications, changes, and variations apparent to those skilled in the art may be made in the arrangement, operation, and details of the methods and systems of the present invention disclosed herein without departing from the scope of the invention, as both described above and claimed below.

WE CLAIM

1. A method, comprising:

Receiving (415) an indication that memory allocation during execution (405) of an application in a managed execution environment (200) exceeds a threshold value; and

Redistributing (430) execution of at least a portion of the application across an isolation boundary.

2. A method according to Claim 1, wherein the method is performed in accordance with a load balancer.

3. A method according to Claim 1, wherein the received indication includes notice to prepare for an anticipated execution time-out.

4. A method according to Claim 1, wherein the received indication includes a notification that an impending sweep of memory heap is expected to cause an execution time-out.

5. A method according to Claim 1, wherein the received indication includes a preview of state in connection with execution of the application.

6. A method according to Claim 1, wherein the receiving further includes receiving an instantiation of an application programming interface (API) from a memory management component associated with the managed execution environment.

7. A method according to Claim 1, wherein the isolation boundary may be any one of a machine boundary, a process boundary, a thread boundary, a class boundary, or an assembly boundary.

8. A method according to Claim 1, wherein the redistributing includes redistributing execution of at least a portion of the application.

9. A method according to Claim 1, wherein the redistributing includes redirecting requests for execution of at least portions of the application across the isolation boundary.

10. A method according to Claim 1, wherein the redistributing includes redirecting a request for execution of a portion of the application to another server.

11. At least one computer-readable medium having one or more executable instructions that, when read, cause one or more processors to:

monitor (410) allocation of memory during execution (405) of managed code;

determine (415) when the memory allocation exceeds a threshold value; and

transmit (425) a notification that the memory allocation has exceeded the threshold value.

12. At least one computer-readable medium according to Claim 11, wherein the one or more executable instructions to transmit cause the one or more processors to transmit the notification to a load balancer.

13. At least one computer-readable medium according to Claim 11, wherein the one or more executable instructions to transmit cause the one or more processors to instantiate an API class.

14. At least one computer-readable medium according to Claim 11, wherein the notification includes an indication that an impending implementation de-allocation of objects may cause the execution of managed code to time-out.

15. At least one computer-readable medium according to Claim 11, wherein the notification includes an instruction to re-direct a request for executing a portion of the managed code across an isolation boundary.

16. At least one computer-readable medium according to Claim 11, wherein the notification includes an instruction to re-direct a request for executing a portion of the managed code across any one of a machine boundary, a process boundary, a thread boundary, a class boundary, or an assembly boundary.

17. At least one computer-readable medium according to Claim 11, wherein the notification includes an instruction to re-direct a request for executing a portion of the managed code to another server.

18. A system, comprising:

a manager (210) to monitor (410) allocation of memory during execution (405) of managed code;

an administrator (215) to send (425) a notification when the monitored allocation exceeds a threshold value; and

a component (315), prompted by the notification, to distribute (430) portions of the managed code for execution across an isolation boundary.

19. A system according to Claim 18, wherein the notification indicates that an impending scan of the memory will time-out the execution of managed code.

20. A system according to Claim 18, wherein the isolation boundary is one of a machine boundary, a process boundary, a thread boundary, a class boundary, or an assembly boundary.