Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2538503 C 2014/05/13

(11)(21) **2 538 503**

(12) BREVET CANADIEN / CANADIAN PATENT

(13) **C**

(22) **Date de dépôt/Filing Date:** 2006/03/01

(41) **Mise à la disp. pub./Open to Public Insp.:** 2006/09/14

(45) **Date de délivrance/Issue Date:** 2014/05/13

(30) **Priorités/Priorities:** 2005/03/14 (US60/662070); 2005/08/31 (US11/216795); 2005/12/22 (US11/317468)

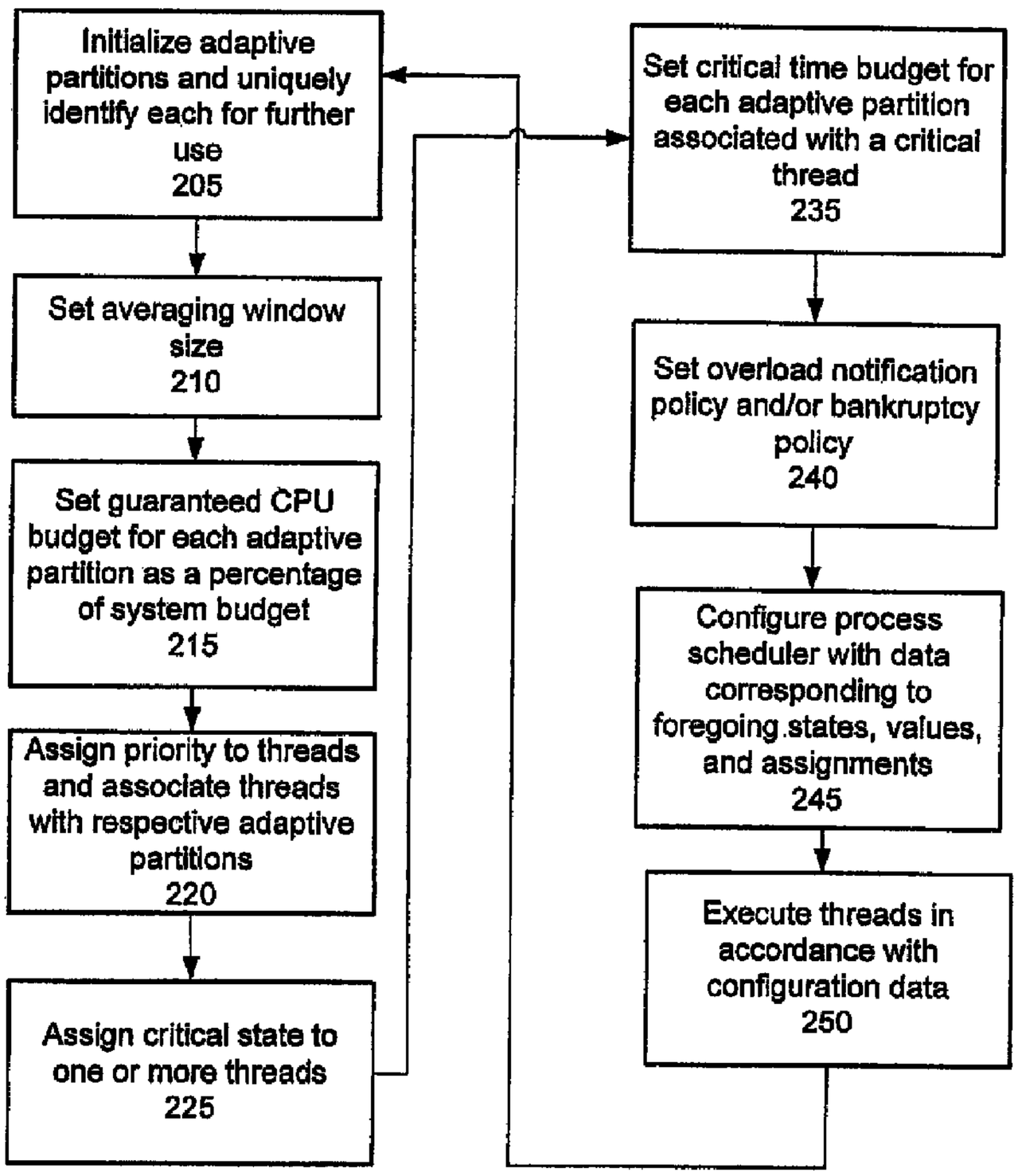
(51) **Cl.Int./Int.Cl.** ***G06F 9/50*** (2006.01), ***G06F 9/38*** (2006.01)

(72) **Inventeurs/Inventors:**
DANKO, ATTILLA, CA;
MARINEAU-MES, SEBASTIEN, CA;
VAN DER VEEN, PETER, CA;
BURGESS, COLIN, CA;
FLETCHER, THOMAS, CA;
STECHER, BRIAN, CA;
DODGE, DAN, CA

(73) **Propriétaire/Owner:**
QNX SOFTWARE SYSTEMS LIMITED, CA

(74) **Agent:** RIDOUT & MAYBEE LLP

(54) **Titre : ORDONNANCEUR DE PROCESSUS A PARTITIONNEMENT ADAPTATIF DES FILES DE PROCESSUS**
(54) **Title: PROCESS SCHEDULER EMPLOYING ADAPTIVE PARTITIONING OF PROCESS THREADS**



(57) **Abrégé/Abstract:**
A system is set forth that comprises a processor, such as a single processor or symmetric multiprocessor, and one or more memory storage units. The system also includes software code that is stored in the memory storage units. The software code is

CA 2538503 C 2014/05/13

(11)(21) **2 538 503**

(13) **C**

(57) **Abrégé(suite)/Abstract(continued):**
executable by the processor and comprises code for generating a plurality of adaptive partitions that are each associated with one or more software threads. Each of the adaptive partitions has a corresponding processor budget. The code also is executable to generate at least one sending thread and at least one receiving thread. The receiving thread responds to communications from the sending thread to execute one or more tasks corresponding to the communications. A scheduling system also forms at least part part of the code that is executable by the processor. In operation, the scheduling system selectively allocates the processor to each sending and receiving thread based, at least in part, on the processor budget of the adaptive partition associated with the respective thread. In this type of sending/receiving environment, the scheduling system bills the processor budget of the adaptive partition associated with the sending thread for processor allocation used by the receiving thread to respond to communications sent by the sending thread.

ABSTRACT

A system is set forth that comprises a processor, such as a single processor or symmetric multiprocessor, and one or more memory storage units. The system also includes software code that is stored in the memory storage units. The software code is executable by the processor and comprises code for generating a plurality of adaptive partitions that are each associated with one or more software threads. Each of the adaptive partitions has a corresponding processor budget. The code also is executable to generate at least one sending thread and at least one receiving thread. The receiving thread responds to communications from the sending thread to execute one or more tasks corresponding to the communications. A scheduling system also forms at least part part of the code that is executable by the processor. In operation, the scheduling system selectively allocates the processor to each sending and receiving thread based, at least in part, on the processor budget of the adaptive partition associated with the respective thread. In this type of sending/receiving environment, the scheduling system bills the processor budget of the adaptive partition associated with the sending thread for processor allocation used by the receiving thread to respond to communications sent by the sending thread.

PROCESS SCHEDULER EMPLOYING ADAPTIVE PARTITIONING OF PROCESS THREADS

INVENTORS:

Dan Dodge
Attilla Danko
Sebastien Marineau-Mes
Peter van der Veen
Colin Burgess
Thomas Fletcher
Brian Stecher

BACKGROUND OF THE INVENTION

1. Technical Field.

[0001] The present invention is directed to a manner in which an operating system schedules the running of threads and the like. More particularly, the invention is directed to an operating system having adaptive partition scheduling for process threads.

2. Related Art.

[0002] The kernel of an operating system should be capable of dividing up CPU resources so that each thread that is active in the system obtains an adequate amount of CPU time to properly execute the corresponding process. To this end, the kernel may implement a scheduling system that determines how the available CPU time is allocated between multiple threads.

[0003] There are at least three types of process scheduling systems: a FIFO scheduling system; a round-robin scheduling system; and a sporadic scheduling system. In each system, a priority value is assigned to each thread of a process that is executed by the CPU. High priority values are assigned to threads that may be important to the operation of the overall system while threads that may be less important to the operation of the system have lower priority values. Whether the scheduling system gives a thread access to the CPU also depends on the state of the thread. For example, a thread may be ready or blocked (although other states also may be used). A thread is ready when it is

capable of being executed in that all conditions needed for it to run have been met. In contrast, a thread is blocked when it tries to initiate an operation that cannot be completed immediately and must wait for the completion of some event before execution of the thread may continue.

[0004] In a FIFO scheduling system, the currently executing thread continues to use all of the CPU time until it gives up the CPU by blocking, it finishes execution, or it is preempted by a higher priority thread. Once one of these criterion are met, the FIFO scheduling system allocates the CPU to the highest priority process/thread that is in a ready state. Generally, there is one ready queue for each priority level.

[0005] A round-robin scheduling system uses an additional parameter, a timeslice, to allocate CPU time to a thread. A timeslice is an amount of time during which a thread is allowed to access the CPU. In a round-robin scheduling system, the currently executing thread continues to use all of the CPU time until the occurrence of one of the following events: 1) the currently executing process blocks; 2) the currently executing process finishes; 3) the currently executing process is preempted by a higher priority thread; or 4) the currently executing process uses up its timeslice. Once the currently executing process blocks or uses up its timeslice, it is put at the back of the ready queue for its priority level.

[0006] Sporadic scheduling is somewhat similar to round-robin scheduling. In a sporadic scheduling system, the currently executing process continues to use all of the CPU time until the occurrence of one of the following events: 1) the currently executing process blocks; 2) the currently executing process finishes; 3) the currently executing process is preempted by a higher priority thread; or 4) the currently executing process uses up a capped limit on the execution time assigned to the thread within a given period of time. The capped limit is known as a budget, while the given period of time in which this budget may be used is known as the replenishment period. In operation, the budget for a thread is replenished upon expiration of the replenishment period. Once the currently executing process blocks, it is put at the back of the ready queue for its priority level. However, if the currently executing process uses up its budget within the replenishment period, its priority level is reduced by a predetermined value and it is placed at the back of the ready queue for this lower priority level. The priority level of

the process/thread may be returned to its original value in response to a number of different conditions.

[0007] In certain operating systems, such as those available from QNX Software Systems in Kanata, Ontario, each thread in the system may run using any of the foregoing scheduling systems. Consequently, the scheduling systems are effective on a per-thread basis for all threads and processes on a node. Each thread is assigned to a particular scheduling system type through the operation of the process/thread itself. This provides the software designer with a significant degree of design flexibility, but also involves a need for coordination between software designers implementing code for the same system. This coordination includes the assignment of priorities to the different threads as well as the scheduling system type assigned to each thread.

[0008] Some available operating systems apply scheduling on a global basis. One such global scheduling system is known as fair-share scheduling. In a fair-share scheduling system, CPU usage may be equally distributed among system users, groups, or processes. For example, if four users (A,B,C,D) are concurrently executing one process each, the fair-share scheduler will logically divide the available CPU cycles such that each user gets 25% of the whole (100% / 4 = 25%). If user B starts a second process, each user will still receive 25% of the total cycles, but both of user B's processes will each receive 12.5% of the total available CPU time. On the other hand, if a new user starts a process on the system, the scheduler will reapportion the available CPU cycles such that each user gets 20% of the whole (100% / 5 = 20%).

[0009] Another layer of abstraction allows partitioning of users into groups, and application of the fair share system to the groups as well. In this case, the available CPU cycles are divided first among the groups, then among the users within the groups, and then among the processes for that user. For example, if there are three groups (1,2,3) containing three, two, and four users respectively, the available CPU cycles may be distributed as follows: 100% / 3 groups = 33.3% per group Group 1: (33.3% / 3 users) = 11.1% per user Group 2: (33.3% / 2 users) = 16.7% per user Group 3: (33.3% / 4 users) = 8.3% per user. Other percentage distributions among the groups also may be used.

[0010] One manner of logically implementing fair-share scheduling strategy is to recursively apply a round-robin scheduling strategy at each level of abstraction (processes, users, groups, etc.). In round robin scheduling, threads of equal importancee

or priority take turns running. They each run for intervals, or timeslices, that are the property of each thread or group of threads.

[0011] While the foregoing scheduling systems have advantages in different applications, they may experience deficiencies when used in certain system applications. For example, when per-thread scheduling systems are used in real-time systems where the latencies of a process/thread have been planned solely through the assignment of priority levels, very long latencies for low-priority threads may occur. Malicious software processes may configure themselves for high priority execution and thereby preempt proper scheduling of lower priority threads. This problem also may occur, for example, during system development when a high priority thread malfunctions and enters an infinite loop. Global fair-share scheduling systems may avoid such problems, but lack the responsiveness needed for use in a real-time system. Therefore, a need exists for a scheduling system that may effectively allow high-priority threads to operate on a real-time basis while concurrently providing some sort of fair-share CPU access to all threads.

SUMMARY

[0012] A system is set forth that comprises a processor, such as a single processor or symmetric multiprocessor, and one or more memory storage units. The system also includes software code that is stored in the memory storage units. The software code is executable by the processor and comprises code for generating a plurality of adaptive partitions that are each associated with one or more software threads. Each of the adaptive partitions has a corresponding processor budget. The code also is executable to generate at least one sending thread and at least one receiving thread. The receiving thread responds to communications from the sending thread to execute one or more tasks corresponding to the communications. A scheduling system also forms at least part part of the code that is executable by the processor. In operation, the scheduling system selectively allocates the processor to each sending and receiving thread based, at least in part, on the processor budget of the adaptive partition associated with the respective thread. In this type of sending/receiving environment, the scheduling system bills the processor budget of the adaptive partition associated with the sending

thread for processor allocation used by the receiving thread to respond to communications sent by the sending thread.

[0013] Other systems, methods, features and advantages of the invention will be, or will become, apparent to one with skill in the art upon examination of the following figures and detailed description. It is intended that all such additional systems, methods, features and advantages be included within this description, be within the scope of the invention, and be protected by the following claims.

BRIEF DESCRIPTION OF THE DRAWINGS

[0014] The invention can be better understood with reference to the following drawings and description. The components in the figures are not necessarily to scale, emphasis instead being placed upon illustrating the principles of the invention. Moreover, in the figures, like referenced numerals designate corresponding parts throughout the different views.

[0015] Figure 1 is a schematic block diagram of one embodiment of a system that may execute a process scheduler in accordance with the teachings of the present invention.

[0016] Figure 2 is a diagram illustrating a number of different interrelated processes that may be used to set up a process scheduler that employs adaptive partitions.

[0017] Figure 3 is a diagram of one exemplary process scheduler that employs a plurality of adaptive partitions with their associated threads and thread priorities.

[0018] Figure 4 is a diagram of a further exemplary process scheduler that employs a plurality of adaptive partitions with their associated threads and thread priorities, where the process scheduler is operating under different conditions than those shown in Figure 3.

[0019] Figure 5 is a diagram of an exemplary process scheduler that employs a plurality of adaptive partitions with their associated threads and thread priorities, where at least one of the ready threads has been designated as a critical thread.

[0020] Figure 6 is a flow diagram showing a plurality of interrelated processes that may be used to implement the process scheduler in a software environment in which there are sending threads and receiving threads.

[0021] Figure 7 is a diagram illustrating the operation of the process scheduler described in connection with Figure 6.

[0022] Figure 8 is a flow diagram showing a plurality of interrelated processes that may be used to schedule the running of threads associated with the various adaptive partitions.

[0023] Figure 9 is a flow diagram showing a plurality of interrelated processes that may be used to determine whether the process scheduler bills the guaranteed budget or critical budget associated with the running thread.

[0024] Figure 10 is a flow diagram showing a plurality of interrelated processes that may be used to calculate the relative budget used by an adaptive partition, where the value of the relative budget used may be employed in the process shown in Figure 8.

[0025] Figure 11 is a flow diagram showing a plurality of interrelated processes that may be used to schedule the running of threads that access the same mutexes.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0026] Figure 1 is a schematic block diagram of a system 100 that may execute a process scheduler employing an adaptive partitioning of process threads. System 100 includes a central processing unit 105 ("CPU") that may access software code stored in memory 110. CPU 105 also may be disposed to access various devices/system components through an I/O interface 115. The devices/system components may include, for example, sensors, human interface devices, network nodes, printers, storage devices, or the like. The processors in a symmetric multiprocessing system, regardless of the number, may be considered in the same manner as a single processor and are likewise included in the scope of the representation by block 105.

[0027] Memory 110 may be used to store, among other things, the software code that defines the functions that are to be executed by the system 100. Although memory 110 is shown as a single unit, it may be implemented as multiple memory units of the same or different memory type. For example, memory 110 may be implemented with

multiple flash memory devices. Alternatively, or in addition, memory 110 may be implemented with one or more flash memory devices and one or more hard disc storage devices. It will be recognized that a substantial number of alternative memory device combinations may be used to implement memory 110.

[0028] Some of the software code that may be stored in memory 110 and executed by CPU 105 is identified in Figure 1. The identified software code shown here includes an operating system 120 and one or more software applications 125. In this example, a process scheduler 130 and device drivers 135 are included as members of the operating system 120. The process scheduler 130 and device drivers 135, however, also may be implemented as software modules that are separate from the operating system 120. Software applications 125 may be used to define the high-level functions that system 100 is to perform. Device drivers 135 may be used to provide a hardware abstraction layer through which software applications 125 communicate with the hardware components of the system 100. The hardware components may include the components accessed through I/O interface 115.

[0029] Process scheduler 130 comprises the executable software code that is used to allocate the processing time of the CPU 105 ("CPU time") to each thread of the system 100. The CPU time may be allocated so that each thread obtains an adequate amount of CPU time to properly execute the corresponding process.

[0030] Figure 2 is a flow chart showing a number of interrelated operations that may be used to implement the process scheduler 130. The flow chart of Figure 2 illustrates many of the values and operational characteristics used in the process scheduler 130. Some values or operational characteristics may be fixed as a result of user choices during the design process, while other values or operational characteristics are provided through configuration data provided by a user and/or through software programming of the system 100.

[0031] As shown, the process scheduler 130 creates a number of different adaptive partitions at block 205. Each adaptive partition constitutes a virtual container for scheduling attributes associated with a set of process threads that have been grouped together by the process scheduler 130. Threads that work toward a common or related function may be assigned to the same adaptive partition. In object-oriented systems, the adaptive partitions may be generated as instances of an adaptive partition class that

includes corresponding scheduling methods and attributes. Each of the adaptive partitions generated at block 205 is uniquely identified for further access by the process scheduler 130.

[0032] The number of partitions generated by the process scheduler 130 may be initiated through the use of configuration parameters. The configuration data may be provided by user initiated commands or through programs that interface with the process scheduler 130.

[0033] At block 210, the size of the averaging window that will be used by the process scheduler 130 is calculated. As will be set forth in further detail below, the averaging window is the time over which the process scheduler 130 attempts to keep adaptive partitions at their guaranteed CPU percentages when the system is overloaded. A typical time is 100 milliseconds, though other averaging window sizes may be appropriate. The averaging window size may be specified at boot time, and may be respecified any time thereafter as well. Considerations in choosing the size of the averaging window may include:

- A short duration averaging window size reduces the accuracy of CPU time-balancing.
- When a short duration averaging window is used, partitions that exhaust their budgets because other partitions are using less than their guaranteed percentage may not have to pay the time back.
- In some instances, a long duration averaging window size may cause some adaptive partitions to experience runtime delays.

[0034] The size of the averaging window ("windowsize") may be assigned in terms of milliseconds, which the scheduling process 130 converts to clock ticks. A clock tick is the interval between clock interrupts (the system timer). All of the time budgets used by the scheduling process 130 may be averaged over the same windowsize.

[0035] At block 215, the scheduling process 130 assigns a guaranteed CPU time budget to each adaptive partition. The guaranteed budget may be assigned as a percentage of the overall available system budget. The sum of all adaptive partitions' CPU percentages in such instances will be 100%. For the purpose of assigning shares of the overall CPU time budget, the processors in a symmetric multiprocessing system, regardless of the number, may be considered in the same manner as a single processor.

[0036] The guaranteed budget used for each adaptive partition may be determined in a number of different manners. For example, the CPU time used by each partition under several different load conditions may be measured and they used to construct a graph of load versus CPU time consumed by each partition. Measurements also may be made under overload conditions. Using this information, a system designer may balance the needs of the various threads contained in each partition under the various conditions and assign the appropriate guaranteed CPU time budgets. The measurements also may be used to dynamically vary the guaranteed CPU time budgets with respect to CPU load conditions. For example, the process scheduler 130 may operate in accordance with different operating modes in response to different operating conditions. While operating in a particular mode, the process scheduler 130 employs a unique set of adaptive partition parameters. The availability and parameters associated with a particular mode may be specified at boot time. A application programming interface ("API") at run-time may be used to switch modes as needed. For example, a first set of guaranteed CPU time percentages may be employed during startup while a second set of guaranteed CPU time percentages may be employed during normal operations after system startup has been completed.

[0037] A priority is assigned to each thread and each thread is associated with a respective adaptive partition at block 220. Functionally-related software may be assigned to the same adaptive partition. This effects a hybrid process scheduling system in which the priority assigned to a thread as well as the guaranteed CPU time percentages of the adaptive partitions are used in the scheduling decisions executed by the process scheduler 130.

[0038] In assigning the threads to respective adaptive partitions, adaptive partition scheduling may be used as a structured way of deciding when the running of a particular function of the system will be inhibited. When used in this manner, separate threads may be placed into different adaptive partitions based on whether the threads should be starved of CPU time under a particular set of circumstances. For example, suppose the system 100 is designed to operate as a packet router. Some of the processes that may be executed by a packet router include 1) routing packets; 2) collecting and logging statistics for packet routing; 3) route-topology protocols with peer routers; and 4) collecting logging and route-topology metrics. In such instances, the threads of these

processes may be assigned to two adaptive partitions: one for threads associated with routing and one for threads associated with the topology of the network. When the system is overloaded, *i.e.*, there is more outstanding work than the machine may possibly accomplish. Consequently, there may be a need to determine which applications are to be run slower. To this end, it may be preferable to route packets as opposed to collecting routing metrics if the CPU does not have enough resources to execute threads for both routing and metrics. It also may be reasonable to run the threads associated with network topology protocols, even when CPU resources are limited. Under these circumstances, it may be preferable to employ three adaptive partitions as opposed to the two adaptive partitions initially considered. The three adaptive partitions, along with exemplary guaranteed budgets, may include:

- an adaptive partition for routing packets (80% share);
- an adaptive partition for topology protocols, (15% share), but with maximum thread priorities higher than the threads in the adaptive partition for routing packets; and
- an adaptive partition, (5% share), for logging of both routing metrics and topology-protocol metrics.

[0039] In this case, adaptive partition scheduling has been used to reorganize various system functions so that certain functions are given priority during high CPU loads, while still ensuring that all system functions are given CPU execution time. Threads associated with routing and threads associated with logging routing metrics have been associated with separate adaptive partitions despite the fact that they are functionally related to routing operations. Similarly, two functionally unrelated components, routing metric logging and topology metric logging, have been associated with the same adaptive partition. This manner of assigning the various threads to different partitions enables the process scheduler 130 to reduce CPU time usage by the logging threads under high CPU load conditions and give priority to routing threads and topology protocol threads while still conducting metrics logging.

[0040] At block 225, a determination may be made as to which threads of the system will be allowed to run in a critical state. Designating a thread as critical gives it the ability to run in a manner that approximates a real-time system, even when the guaranteed budget for the adaptive partition budget might otherwise be exceeded. When

a critical thread associated with a particular adaptive partition is run despite the lack of guaranteed budget for the adaptive partition, the adaptive partition is said to have gone into short-term debt. Critical threads may be associated with the various partitions, for example, at boot time. Critical threads are discussed in further detail below.

[0041] Each adaptive partition that is associated with a critical thread is assigned a critical time budget at block 235. The critical time budget may be specified, for example, in time units, such as milliseconds. In the exemplary system, the critical time budget is the amount of CPU time available to all critical threads associated with a given adaptive partition above that partition's guaranteed time budget during an averaging window. By employing critical designations and critical time budgets, a critical thread generally has an opportunity to run on the CPU even if its associated adaptive partition has exhausted its guaranteed budget. This will occur as long as the partition still has critical time budget available. Critical threads may provide the ability for real-time behavior within these partitions.

[0042] Various policies that the process scheduler 130 must follow may be optionally assigned to the system 100 at block 240. For example, a bankruptcy policy may be applied to one or more of the adaptive partitions to determine how the system 100 and/or process scheduler 130 handles a bankrupt state of the adaptive partition. Bankruptcy of a critical adaptive partition occurs when the adaptive partition has exhausted both its guaranteed CPU time budget and critical time budget over the duration of the averaging window. As a further example, system 100 may execute an overload notification policy that allows a process to register with the process scheduler 130 so that it is notified when a system overload occurs. A system overload occurs, for example, when all of the ready threads cannot be executed on the CPU over the duration of the averaging window. A process may register to be informed of an overload condition when the system 100 enters and/or leaves the overload state. Applications may use this overload notification to gracefully degrade their service, for example, by skipping less important functions, or by reducing the precision of computation. Adaptive partitions may go over budget when some other adaptive partition is sleeping, as will be set forth below. This is not by itself necessarily considered to be a system overload, and therefore does not require triggering of the overload-notification policy.

[0043] At block 245, the process scheduler 130 is configured with data corresponding to the foregoing states, values and/or assignments provided at blocks 205 through 240. As noted above, these states, values and/or assignments may be provided for use in the system 100 in a number of different manners, such as, by the system designer, by the system user, through other software programs, etc.

[0044] Block 250 represents execution of the threads in accordance with the configuration data of block 245. The attributes of the various adaptive partitions may be dynamic. To this end, the parameters set forth in one or more of the foregoing blocks may be changed in response to system requirements, system state, changes in system functions, etc., as represented by the flowline returning to block 205.

[0045] The threads in system 100 also may vary dynamically over time. For example, a thread or group of threads associated with an adaptive partition may spawn, or generate, other threads during operation. The originating thread may be referred to as a parent thread while a spawned thread may be referred to as a child thread. Process scheduler 130 may be configured so that child threads inherit the adaptive partition of their parent thread automatically. Alternatively, or in addition, an API may be provided that will allow spawning threads into other adaptive partitions. Such an API may be made available only to code with sufficient privilege. For example, a system application launcher may have such privileges.

[0046] Figure 3 illustrates one manner in which the process scheduler 130 may operate under normal load conditions in which none of the partitions exceeds its CPU budget. In this example, the process scheduler 130 has generated three adaptive partitions 301, 302 and 303. Adaptive partition 301 may be associated with the threads 305, 306, 307, and 308, of a multimedia application. Adaptive partition 302 may be associated with the threads 315, 316, 317, 318, and 319 of a Java application. Adaptive partition 303 may be associated with the threads 320, 325, 330, 335, and 340 of a system logging application. The threads 305-345 have various scheduling priorities, denoted in parentheses in Figure 3, that may be independent of the guaranteed budget of the associated adaptive partition.

[0047] In operation, each adaptive partition 301-303 and thread 305-345 may assume different operative states. Adaptive partitions 301-303, for example, may operate in an active state or a sleep state. In the active state, the scheduling attributes of

the adaptive partition may be used to schedule CPU time for the associated threads. A sleep state occurs when there are no ready threads associated with the adaptive partition. In such instances, the process scheduler 130 effectively treats the adaptive partition as non-existent.

[0048] Threads may assume, for example, a running state, a ready state or a blocked state. A thread is in the running state while it is being executed by the CPU. It is in a ready state when a set of conditions have been met that render the thread fully prepared for execution by the CPU at a time determined by the process scheduler 130. A thread is in the blocked state while the thread waits for the occurrence of one or more events. While in the blocked state, the thread is not ready to consume any CPU resources. Once the events awaited by the thread occur, the thread may become unblocked and enter the ready or running state.

[0049] The adaptive partitions 301-303 and corresponding threads 305-345 can be used to describe the operation of the process scheduler 130 during various load conditions. In this example, the process scheduler 130 makes the CPU available to execute ready threads 308, 315, and 345 assigned to each adaptive partition 301, 302, and 303, based on the priority of the ready threads. Under normal load conditions, the highest-priority thread in the system 100 will run immediately when it becomes ready. Whether a thread is ready may be indicated to the process scheduler 130 in a number of different manners including, for example, through the occurrence of an interrupt event or the like. In the illustrated example, the highest priority ready thread is thread 345, which has a priority of 17. Thread 345 will continue to operate in a running state until it is finished, blocked, or until the budget for adaptive partition 303 is exhausted. Under heavy load, if an adaptive partition exceeds its CPU budget, then its highest-priority thread does not run until the partition once again has time available in its CPU budget. This is a safeguard on the system 100 that divides insufficient CPU time among the partitions 301, 302, and 303. In this state, the processor runs the highest-priority thread in an adaptive partition with CPU time remaining in its guaranteed CPU time budget.

[0050] When an adaptive partition enters a sleep state, the process scheduler 130 allocates the CPU budget of the sleeping partition to other active adaptive partitions — even if the other active adaptive partitions have exceeded their budget. For example, if adaptive partition 303 enters a sleep state, the process scheduler 130 allocates the budget

for adaptive partition 303 to adaptive partition 302, since adaptive partition 302 has the highest priority ready thread 315. If two or more adaptive partitions have threads with the same highest priority, the process scheduler 130 divides the free time in proportion to the other adaptive partitions' percentages. This allocation assists in preventing long ready-queue delay times in the case where two adaptive partitions have the same priority.

[0051] In the example of Figure 4, there are three adaptive partitions 401, 402, and 403, with 70%, 20% and 10% CPU budget guarantees, respectively. Further, each adaptive partition 401, 402, and 403, includes a ready thread 408, 415, and 445, having a priority of 14. If adaptive partition 401 enters a sleep state through a blocking of thread 408, the process scheduler 130 allocates all of the available CPU time to adaptive partitions 402 and 403 in a 2:1 ratio, the ratio corresponding to their original CPU budget allocations. If adaptive partition 401 is in a sleep state for a short time, then the process scheduler 130 may ensure that partition 401 later receives CPU time at its guaranteed CPU time budget by reallocating the CPU resources so that adaptive partitions 402 and 403 pay back the CPU time that each utilized at the expense of partition 401. If adaptive partition 401 is in a sleep state for a long time, then some or all of the time used by adaptive partitions 402 and 403 may become free. Whether an adaptive partition is in a sleep state for a short time or a long time can be determined in various manners. For example, an adaptive partition can be said to be in a sleep state for a short time when it is in the sleep state for a duration of time that is less than (windowsize) - (budget percentage*windowsize) milliseconds within one averaging window.

[0052] If all adaptive partitions are at their CPU budget limits, then the process scheduler 130 may specify running of the highest-priority thread in the system 100. If two adaptive partitions have threads with the same highest priority, then the adaptive partition that has used the smallest fraction of its budget may be run. This manner of operation may be used to prevent long ready-queue delays that would otherwise occur. In the example shown in Figure 4, the window size may be 100 ms, adaptive partition 401 is allotted 70% of the CPU budget and has used 40 ms, adaptive partition 402 is allotted 20% of the CPU budget and has used 5 ms, and adaptive partition 403 is allotted 10% of the CPU budget and has used 7 ms. All partitions have a ready thread at priority 14. In this situation, thread 415 of adaptive partition 402 is run because its relative

fraction free is 5 ms/20 ms, or 0.25, while the relative fraction free for adaptive partition 401 is 40 ms/70 ms or 0.57 and 7 ms/10 ms or .70 for adaptive partition 403.

[0053] If an adaptive partition has exhausted the assigned CPU budget and one or more of its ready threads are designated as critical, then the process scheduler 130 may allow the adaptive partition to use its critical CPU budget to run the critical threads. The critical time budget is the amount of CPU time that the process scheduler 130 allocates to an adaptive partition to run all critical threads associated with the adaptive partition. This critical time budget constitutes CPU time that the process scheduler 130 allocates to the adaptive partition the partition's normal budget during an averaging window. Consequently, a critical thread may run even if the adaptive partition with which it is associated is out of budget, as long as the adaptive partition has not exhausted its critical time budget.

[0054] Figure 5 illustrates a system having two adaptive partitions 501 and 502. Threads 505, 506, 507, and 508 are associated with adaptive partition 501, while threads 515, 516, 517, 518, and 519 are associated with adaptive partition 502. Adaptive partition 501 may comprise threads that are associated with, for example, an airbag notification process. Threads 505 and 508 of adaptive partition 501 half been designated as critical threads. However, only thread 508 is in a ready state.

[0055] In this example, adaptive partition 501 has exhausted its guaranteed CPU time budget. Nonetheless, critical thread 508 may be allowed to run because of the importance that has been placed on it through the critical designation. Consequently, the process scheduler 130 may allocate CPU time to critical thread 508 provided there is still CPU time in the critical time budget of adaptive partition 501.

[0056] When the process scheduler 130 schedules the running of a critical thread, such as thread 508, the process scheduler bills the thread's run time against the available guaranteed CPU time budget of the thread's associated adaptive partition. However, the process scheduler 130 counts the CPU time used by the critical thread against the critical time budget of the adaptive partition only if the process scheduler 130 would not have otherwise run the critical thread. To this end, CPU time used in running a critical thread is not counted against the critical time budget of the associated adaptive partition when 1) the system is not overloaded, or 2) the system is overloaded, but one or more adaptive partitions may not be exhausting their guaranteed CPU time budgets.

[0057] A critical thread may remain in a critical running state until it enters a blocking state. That is, it may leave the running or ready state as is the case with any other thread. As noted above, this may occur while the thread is waiting for a message, interrupt notification, etc. The criticality of a thread, or billing to its adaptive partition's critical time budget, may be inherited along with the adaptive partition during operations which trigger priority inheritance.

[0058] The short-term debt is bounded by the critical time budget specified for the partition. Over time, the partition may be required to repay short-term debt. A critical thread that exceeds the critical time budget of the adaptive partition may be considered to drive the associated adaptive partition into bankruptcy. Bankruptcy may be handled as an application error, and the designer may specify the system's response. Exemplary choices for a response include: 1) forcing the system to reboot; 2) notifying an internal or external system watchdog; and/or 3) terminating and/or notifying other designated processes. The system may add an entry to a log or the like if an adaptive partition exhausts its critical time budget. When the budgets for the adaptive partitions are dynamically altered (through, for example, a mode change, an API call to modify CPU percentages, or the like), the process scheduler 130 may impose a delay before testing an adaptive partition for bankruptcy. This allows the budgets to stabilize before an adaptive partition may be declared bankrupt.

[0059] The designation of a thread as critical may occur in a number of different manners. For example, the system may automatically mark threads that are initiated by an I/O interrupt as critical. The system also may specify a set of additional applications or operating system notification events, for example, timers, which may mark their associated handler threads as critical. An API also may be used to mark selected threads as critical. Still further, child threads of critical parent threads may be automatically designated as critical.

[0060] The process scheduler 130 also may be adapted for use in client/server systems in which messages are passed from one or more sending/client threads for receipt and/or processing by one or more receiving/server threads. Figure 6 shows a number of interrelated processes that may be used to implement process scheduler 130 in such a client/server environment. At block 605, a client thread in a first adaptive partition sends a message to a server thread in a second adaptive partition. When a client

thread sends a message to a server thread, the server thread that receives the message may inherit the priority of the sending thread. This prevents priority inversion, since the server thread is doing work on behalf of the sending client. Inheriting of the priority level of the sending thread by the server thread is shown at block 610. The process scheduler also may assign the same critical status of the client thread to the server thread at block 610. At block 615, the process scheduler 130 may associate the server thread with the same adaptive partition that is associated with the client thread.

[0061] Figure 7 illustrates a system that may be used to explain some aspects of the operations shown in Figure 6. In this example, there are three adaptive partitions 701, 702, and 703. Adaptive partition 702 may include server threads 720 and 725 of the type run, for example, by the processes of a file system. The attributes assigned to adaptive partition 702 may vary with the design of the system. However, in this example, adaptive partition 702 has been generated with a CPU time budget of zero, and threads 720 and 725 have been assigned a default priority of 7. Adaptive partition 701 may comprise a number of different threads 705, 710, and 715, where thread 715 constitutes a client thread that sends messages to one or more server threads of adaptive partition 702. Similarly, adaptive partition 703 may comprise a number of different threads 730, 735, 740, and 745, where thread 745 constitutes a client thread that sends messages to one or more server threads of adaptive partition 702.

[0062] As shown by line 750 of Figure 7, client thread 715 has passed a message that is received or otherwise processed by thread 720 of adaptive partition 702. Similarly, client thread 745 of adaptive partition 703 has passed a message that is received or otherwise processed by thread 725 of adaptive partition 702, as indicated by line 755. When the message transfers 750 and 755 occur, the process scheduler 130 associates each server/receiving with the adaptive partition of the thread that sent the message. In this case, server thread 720 is temporarily associated with adaptive partition 701, as indicated by line 760, and is assigned the same priority, 14, as thread 715. Likewise, server thread 725 is temporarily associated with adaptive partition 703, as indicated by line 765, and is assigned to the same priority, 12, as thread 745. Both threads 720 and 725 may be placed in a ready state, while threads 715 and 745 may be placed in a blocked state as the respective messages are processed. Once thread 720 has completed processing the message received from thread 715, thread 720 may return to its

original state where it is associated with adaptive partition 702 with a default priority of 7. Also, thread 725 may return to its original state where it is associated with adaptive partition 702 with a default priority of 7 once it has completed processing of the message received from thread 745.

[0063] Sometimes, a client thread may attempt to communicate with a server thread that is busy processing one or more messages that have been previously received from other client threads. For example, if thread 740 of adaptive partition 703 becomes unblocked and attempts to send a message to server thread 725 while server thread 725 is busy processing one or more prior messages received from thread 745, server thread 725 typically will be unable to respond to thread 740 until it has completed processing the prior messages from client thread 745. In such instances, the process scheduler 130 temporarily may raise the priorities of one or more server threads that, for example, wait on the same connection path as the client thread. The query and action corresponding to these operations are illustrated at blocks 620 and 625 of Figure 6. As applied to the system of Figure 7, the connection path between threads assigned to adaptive partition 703 and threads assigned to adaptive partition 702, for example, may be assigned the same identification name/number. As a result, the priority of server thread 725 may be raised in an attempt to reduce the latency that may otherwise occur before server thread 725 is available to process the message from client thread 740. Depending on the nature of the threads in adaptive partition 702, the priority levels of multiple server threads may be raised.

[0064] At block 630 of Figure 6, the process scheduler 130 operates to bill the execution time for each thread 720 and 725 in the appropriate manner. The appropriate manner of billing the execution time may vary. One manner includes applying the execution time of the receiving thread against the CPU budget and/or critical CPU budget of the adaptive partition associated with the sending thread. In the example of Figure 7, the execution time used by thread 720 in responding to a message sent by thread 715 is counted against the CPU budget and/or critical CPU budget of adaptive partition 701. Similarly, the execution time used by thread 725 in responding to a message sent by thread 745 is counted against the CPU budget and/or critical CPU budget of adaptive partition 703.

[0065] System components, such as filesystems, device drivers, and the like may be assigned a guaranteed budget of zero. In such instances, the CPU time used by the threads associated with the system component is billed to their clients. However, sometimes the process scheduler 130 finds out too late which threads a particular system component thread has been ultimately working for. As a result, the process scheduler 130 may not be able to bill the running of the threads of the system components in a timely manner and/or to the proper adaptive partition. Additionally, some system components, such as device drivers, may have background threads (*e.g.* for audits or maintenance) that require budgets that cannot be attributed to a particular client. In those cases, the system designer may measure the background operations and unattributable loads associated with the various system components. The resulting measurements may be used to provide non-zero budgets to the adaptive partitions associated with the threads of the various system components.

[0066] In operation, the process scheduler 130 may do more than simply direct the running of the highest priority ready thread associated with an adaptive partition having guaranteed CPU time left in its budget. For example, when all adaptive partitions have exhausted their guaranteed CPU time budgets at approximately the same time, then the process scheduler 130 may direct the running of the highest priority thread in the system irrespective of the attributes of the associated adaptive partition. Also, when adaptive partitions have threads of the equal highest priorities, the process scheduler 130 may assign CPU time using the ratio of their guaranteed CPU time percentages. Finally, critical threads may be run even if their adaptive partition is out of budget, provided the adaptive partition still possesses an amount of its critical time budget.

[0067] Process scheduler 130 may employ one or more ordering functions, f(ap), associated with each adaptive partition, ap, in its scheduling determinations. Ordering functions may be calculated in a variety of different manners. The results obtained by calculating the ordering functions may be compared with one another to determine how the process scheduler 130 will scheduled the threads associated with the various adaptive partitions of the system.

[0068] In calculating the ordering functions, f(ap), a number of different variables may be used. A view of the variables are described here in connection with one example. In the following example, let "is_critcal(ap)" be a boolean variable. The

value of "is_critcal(ap)" depends on 1) whether the adaptive partition, ap, has available critical budget, and 2) whether the highest priority ready thread in the adaptive partition, ap, has been assigned a critical state. Let "has_budget(ap)" be a boolean variable that indicates whether an adaptive partition, ap, has consumed less CPU time than its guaranteed CPU time budget during the last averaging window. Let "highest_prio(ap)" be an integer variable that indicates the highest priority of all ready-to-run threads in an adaptive partition, ap. Let "relative_fraction_used(ap)" be a real number variable that corresponds to the ratio of the number of microseconds of CPU time consumed by the adaptive partition, ap, during the last averaging window, divided by the budget of the adaptive partition, ap, when expressed, for example, in microseconds. Finally, let b(ap) be a boolean variable corresponding to the current rate of CPU time consumption of threads in the adaptive partition, ap. More particularly, b(ap) may be assigned a logical true value if, given the current rate of CPU time consumption by the threads associated with the partition, ap, the value of has_budget(ap) also would be a logical true value at the time the process scheduler 130 would likely be called upon to again schedule a thread associated with the adaptive partition. Otherwise, b(ap) may be assigned a logical false value. It will be recognized that other variables, or less than all of the foregoing variables, may be used to calculate and ordering function f(ap). Which variables are used may be dependent on the system designer and/or end-user.

[0069] The value of b(ap) may be calculated in a number of different manners. For example, let the variable t indicate the current time in a high-resolution counter, and a tick be the length of time between regular events during which the process scheduler examines all partitions. The period of the tick should be less than the size of the averaging window ("windowsize"). Let the function cpu_time_used(t1,t2) correspond to a real value of the CPU time used by the threads of partition, ap, between absolute times t1 and t2. Further, let budget(ap) correspond to the time value of the guaranteed budget for the partition, ap. The value for b(ap) then may be calculated using the following equation:

[0070] b(ap) = Boolean (cpu_time_used(now, now-windowsize) - cpu_time_used(now-windowsize+tick, now-windowsize) < budget(ap)).

[0071] One manner of calculating an ordering function, f(ap) using the foregoing variables is shown in Figure 8. The illustrated operations may be executed for each adaptive partition, ap, used in the system 100. As shown, the process scheduler 130 determines whether the partition has any threads that are ready to run at block 800. If there are no ready threads associated with the adaptive partition, ap, the ordering function, f(ap), for the adaptive partition may be assigned the values f(0,0,0,0), at block 802 and calculation of the ordering function for the next adaptive partition may be initiated at block 803. The process scheduler 130 determines if the adaptive partition, ap, has critical CPU budget available at block 805 and, if so, whether the highest priority ready thread in the adaptive partition, ap, has been assigned a critical state. Based on this determination, the process scheduler 130 assigns the appropriate logical state to the "is_ critical(ap)" variable at block 810. At block 813, the process scheduler 130 evaluates the CPU budget used by the adaptive partition during the last averaging window. At block 815, the process scheduler 130 determines if the adaptive partition has used less CPU time than its guaranteed CPU time budget. Based on this determination, the process scheduler 130 assigns the appropriate logical state to the "has_budget(ap)" variable at block 820. The relative ratio of CPU budget time used during the last averaging window is calculated at block 830 by taking the value obtained at block 813 and dividing it by the guaranteed CPU budget time for that adaptive partition. This value is assigned to the "relative_fraction_used(ap)" variable at block 835.

[0072] At block 840, the process scheduler 130 calculates one or more values corresponding to the current rate of CPU time consumption by threads associated with the adaptive partition. These values are used to assign the appropriate logical state to the b(ap) variable at block 845.

[0073] Using all or a subset of the foregoing variables, the ordering function, f(ap), for the given adaptive partition, ap, is calculated at block 850. In this example, the ordering function, f(ap), is calculated using the ordered values (x,a,y,z), where x=is_critical(ap) OR has_budget(ap); a=(Not x) AND b(ap); y=highest_prio(ap); and z=1-relative_fraction_used(ap). In calculating the ordering function, f(ap), the value of x is given more significance than the values of a, y, or z, the value of a is given more

significance than the values of y or z, and the value of y is given more significance than the value of z.

[0074] The process scheduler 130 runs the highest priority thread of the adaptive partition having the largest f(ap) as shown at block 905 of Figure 9. However, the process scheduler 130 must determine whether the running time for the thread is to be billed to the critical budget of the adaptive partition or solely to the guaranteed budget. To this end, the process scheduler 130 may compute another function, fcritical(ap), using one or more of the foregoing variables. In this example, fcritical(ap) is calculated using the ordered values (w,d,y,z), where w=has_ budget(ap); d=(NOT w) AND b(ap); y= highest_prio(ap); and z= 1-relative_fraction_used(ap). In the calculation of fcritical(ap), the value of w is given more significance than the values of d, y, or z, the value of d is given more significance than the values of y or z, and the value of y is given more significance than the value of z. This calculation is shown at block 910. A comparison between the fcritical(ap) and f(ap) values for the adaptive partitions is executed at block 915. If the value of fcritical(ap) is less than the value of f(ap), then the running time for the thread is billed to the critical budget of the adaptive partition at block 920 as well as to the guaranteed budget at block 925. If the value of fcritical(ap) is greater than or equal to the value of f(ap), then the running time for the thread is solely billed to the guaranteed budget of the adaptive partition at block 925. Any calculations used by the process scheduler 130 should ensure that the critical budget for the adaptive partition is only used if the highest priority critical thread associated with that adaptive partition would not have been selected to run by process scheduler 130 had the thread or the partition not been critical.

[0075] Figure 10 shows a number of different interrelated processes that may be used to execute a method for calculating the relative ratio of CPU budget time consumed during the last averaging window, as described in block 830 of Figure 8. The relative fraction CPU budget time consumed by an adaptive partition is, for example, the number of microseconds that threads associated with the adaptive partition ran during the last averaging window divided by its share of the averaging window in microseconds. Alternatively, this may be expressed as (total_CPU_time_consumed) /(windowsize*percentage). To reduce the potential need for floating-point division, the process scheduler 130 may compute a number that has substantially the same ordering

properties as (total_CPU_time_consumed)/(windowsize*percentage) for each adaptive partition. Thus, a constant c(a) may be pre-computed so that the adaptive partition with the highest (total_CPU_time_consumed)*c(ap) also is the adaptive partition with the highest (total_CPU_time_consumed)/(windowsize*percentage).

[0076] The function c(ap) may be precalculated during, for example, system startup, and whenever the guaranteed CPU budgets are reallocated between the various adaptive partitions of the system. At block 1010, the CPU budget percentage for each adaptive partition are determined, for example, at start-up. At block 1015, the system may compute, for each adaptive partition, a factor, q(ap). The value of q(ap) may be calculated, for example, as the product of the percentage CPU budgets of all the other adaptive partitions. At block 1020, a scaling factor is calculated. In this example, if the maximum averaging error is max_error (*e.g.* 0.005 for 1/2 a percent), then k = min(list of q(ap)) * max_error. A constant scaling factor c(ap) is calculated at step 1025. In this example, c(ap) is calculated as c(ap)=q(ap)/k. The value, (total_CPU_time_consumed)*c(ap) has the same ordering properties as (total_CPU_time_consumed)/(windowsize*percentage) within an error tolerance of max_error.

[0077] To practically compare the relative fraction used by different adaptive partitions, the process scheduler 130 may need to multiply the run time of the threads associated with the adaptive partitions by c(ap). However the billed times may be large numbers. If the process scheduler 130 is to be implemented using single-multiply instructions in these calculations, the billed times may be first scaled choosing a number of most significant bits of the CPU budget time at block 1030. The degree of scaling may be set by the value of max_error. However, any reasonable choice for max_error (*e.g.* 1/4 to 1/2 %) can be satisfied by choosing only the most significant 16 bits of the billed run-time. In such instances, the system may be calculating (total_CPU_time_consumed)>>32 * c(ap). At block 1035, the relative budget ratio is calculated as c(ap)*(adaptive partition execution time), where (adaptive partition execution time) constitutes a selected number of the most significant bits of (total_CPU_time_consumed).

[0078] An error tolerance of 0.5% to 0.25% is considered sufficient for an implementation. However, the application may include the notion that for any specified

error tolerance, a minimal number of bits is chosen to both represent c(ap), the scaled value of the CPU time executed by adaptive partition ap, during the last averaging windowsize time, and the product of c(ap) and the scaled CPU time. The minimal number of bits is chosen for both representations and executing multiplication functions so that all representations and arithmetic errors are less than or equal to a chosen error tolerance.

[0079] Billing of CPU time to each of the adaptive partition in a system may take place in a number of different manners and may occur many times during the operation of the process scheduler 130. For example, billing of an adaptive partition may occur whenever 1) a thread starts running from a blocked state, 2) a thread stops running (i.e., when it has been preempted by a higher priority thread, when it has been blocked, or the like), and/or 3) at other times when an accurate accounting of the CPU time is needed by the process scheduler 130.

[0080] Typically, process schedulers use standard timer interrupts, or ticks, to determine how long a thread has used the CPU. Tick periods are often on the order of one to several milliseconds.

[0081] The process scheduler 130, however, may include code that effectively microbills the execution time of the various threads of the system. To this end, a high-resolution hardware and/or software counter having a period substantially less than the tick periods may be employed. Each time a thread starts or stops running, the process scheduler 130 assigns a timestamp to the associated partition corresponding to the value of the high-resolution counter. The timestamp values may be scaled to a useful common time unit. The differences between the timestamps for adjacent start and stop times of a thread are used to microbill the appropriate adaptive partition.

[0082] The high-resolution counter may be implemented in a number of different manners. For example, some CPUs have a built-in counter that increments at about the clock frequency at which the CPU is run. In such situations, the built-in counter may be used in the microbilling process. In another example, a high-resolution counter may be simulated using software by querying an intermediate state of a programmable count-down timer that, for example, may normally be used to trigger clock interrupts. This may be the same counter used to provide an indication that a tick interval has occurred. In such situations, the timestamps should take into consideration both the counter value

and the number of ticks that have occurred from a given reference point in time so that the timestamps accurately reflect the start times and stop times of the individual threads.

[0083] The foregoing process scheduler 130 also may be used in systems that employ mutexes. Mutexes are used to prevent data inconsistencies due to race conditions. A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depend on the order in which these operations are performed. Mutexes may be used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource may have a mutex associated with it. One may apply a mutex to protect a segment of memory ("critical region") from other threads. The application gives a mutex to threads in the order that they are requested. However, the process scheduler 130 may be adapted to deal with the problems that occur when a low-priority thread, which may hold the mutex, unreasonably delays access to higher-priority threads that are waiting for the same mutex.

[0084] Figure 11 shows a method for prioritizing access to a mutex in an adaptive partitioning system, when one thread may hold a mutex, and several other threads may be waiting for the same mutex. When the partition associated with the thread holding the mutex runs out of guaranteed CPU time budget, the process scheduler 130 may begin billing the run time of the thread holding the mutex to the partition of the thread waiting for the mutex which, of all the threads waiting for the mutex, is most likely to run next. The process scheduler 130 also may begin billing the run time of any thread deemed to be working on behalf of the thread holding the mutex to the adaptive partition associated with the thread holding the mutex. A determination is made at block 1110 to identify the thread that is most likely to run next (i.e., the thread that will run after the thread holding the mutex is blocked, finishes, or the like). The waiting thread, which may be waiting for the same mutex as the current thread holding the mutex, may be determined to be "most likely to run next". At block 1115, the process scheduler 130 may raise the priority level of the thread holding the mutex to the priority of the waiting thread which is most likely, of all the waiting threads, to run next. The process scheduler 130 may bill the adaptive partition associated with the current thread holding the mutex for its running time while holding the mutex until the adaptive partition's CPU budget is exhausted (reaches zero) at block 1120. At block 1125, the remaining CPU time used by

the holding thread is billed to the adaptive partition that is associated with the waiting thread that is most likely to run next.

[0085] The thread "most likely to run next" may be computed by applying, pairwise, a "compare two threads" process repeatedly on pairs of threads in a list of waiting threads. The "compare two threads" process may be executed as follows, where A and B are the two threads to be compared: A function f(ap) is constructed, which includes the ordered values (x,a,y,z). This is the same ordering function f(ap) constructed above. Then, let partition_of(X) mean the partition containing the thread X. Then, if f(partition_of(A)) > f(partition_of(B), thread A is more likely to run than thread B. The function f(X) is constructed for each thread to be compared until the thread with the highest f(X) is determined. The thread with the highest f(X) may be determined to be the "thread most likely to run next" and its associated adaptive partition may be billed accordingly for the running time of the thread holding the mutex once the adaptive partition associated with the thread holding the mutex has exhausted its guranteed CPU budget.

[0086] The systems and methods described above may be configured to run in a transaction processing system where it is more important to continue to process some fraction of the offered load rather than to fail completely in the event of an overload of processing capacity of the system. Examples of such applications include Internet routers and telephone switches. The systems and methods also may be configured to run in other real-time operating system environments, such as automotive and aerospace environments, where critical processes may be designated that need to be executed during critical events. An example may be in an automotive environment, where an airbag deployment event is a low probability event, but must be allocated processor budget should the event be initiated.

[0087] The systems and methods also may be configured to operate in an environment where untrusted applications may be in use. In such situations, applications such as Java applets may be downloaded to execute in the operating system, but the nature of the application may allow the untrusted application to take over the system and create an infinite loop. The operating system designer will not want such a situation, and may create appropriate adaptive partitions so the untrusted application may be run in isolation, while limiting access to CPU time which other processes will have need of.

[0088] While various embodiments of the invention have been described, it will be apparent to those of ordinary skill in the art that many more embodiments and implementations are possible within the scope of the invention. Accordingly, the invention is not to be restricted except in light of the attached claims and their equivalents.

Claims:

1. A system comprising:

a processor;

one or more memory storage units;

software code stored in the one or more memory storage units, wherein the software code is executable by the processor and comprises:

a plurality of adaptive partitions each having an associated guaranteed processor time budget and one or more associated process threads from a plurality of process threads;

the plurality of process threads each having a priority and each dynamically associable with any one of the plurality of adaptive partitions; and

a process scheduler executable by the processor configured to:

when the system is under a normal load, allocate the processor to a process thread, of the plurality of process threads, that is in a ready state and having the highest priority amongst process threads, of the plurality of process threads, that are in the ready state; and

when the system is in overload, allocate the processor to a process thread, of the plurality of process threads, that is in the ready state, having the highest priority amongst process threads, of the plurality of process threads, that are in the ready state and for which the adaptive partition that the process thread is associated with has available guaranteed processor time budget;

wherein the guaranteed processor time budget is allocated and billed over successive averaging windows; and

wherein a portion of the guaranteed processor time budget associated with each adaptive partition is billed whenever the processor is allocated to a process thread associated with the adaptive partition.

2. The system of claim 1, the plurality of process threads comprising a sending thread that sends a message and a receiving thread that receives the message;

wherein the guaranteed processor time budget associated the adaptive partition with which the sending thread is associated is billed when the processor is allocated to the receiving thread to process the message.

3. The system of claim 2, wherein the receiving thread is temporarily associated with the adaptive partition with which the sending thread is associated while the receiving thread is processing the message.

4. The system of claim 1, wherein the system is in overload when all of the process threads in the ready state cannot be allocated the processor over the duration of one averaging window, of the successive averaging windows, and wherein the system is under normal load otherwise.

5. The system of claim 1, wherein the guaranteed processor time budget associated with each of the plurality of adaptive partitions is dynamic and can be reallocated while the system executes.

6. The system of claim 1, wherein the process thread to which the processor is allocated runs until the process thread is finished, is blocked, or the guaranteed processor time budget associated with adaptive partition with which the process thread is associated is exhausted.

7. The system of claim 6, wherein the process thread to which the processor is allocated further runs until the process thread is preempted and the processor is allocated to another process thread, of the plurality of process threads, that is in the ready state and that has a priority higher than the priority of the process thread to which the processor is currently allocated.

8. A method of scheduling a plurality of process threads, each having a priority and each dynamically associable with an adaptive partition, for execution by a processor of a system, the method comprising:

creating a plurality of adaptive partitions each having an associated guaranteed processor time budget;

associating one or more process threads from the plurality of process threads with each of the plurality of adaptive partitions;

when the processor is under a normal load, allocating the processor to a process thread, of the plurality of process threads, that is in a ready state and having the highest priority amongst process threads, of the plurality of process threads, that are in the ready state; and

when the processor is in overload, allocating the processor to a process thread, of the plurality of process threads, that is in the ready state, having the highest priority amongst process threads, of the plurality of process threads, that are in the ready state and for which the adaptive partition that the process thread is associated with has available guaranteed processor time budget;

wherein the guaranteed processor time budget is allocated and billed over successive averaging windows; and wherein a portion of the guaranteed processor time budget associated with each adaptive partition is billed whenever the processor is allocated to a process thread associated with the adaptive partition.

9. The method of claim 8, further comprising:

sending a message from a sending thread of the plurality of process threads;

receiving the message by a receiving thread of the plurality of process threads; and

billing the guaranteed processor time budget associated the adaptive partition with which the sending thread is associated when the processor is allocated to the receiving thread to process the message.

10. The method of claim 9, further comprising:

temporarily associating the receiving thread with the adaptive partition with which the sending thread is associated while the receiving thread is processing the message.

11. The method of claim 8, wherein the system is in overload when all of the process threads in the ready state cannot be allocated the processor over the duration of one averaging window, of the successive averaging windows, and wherein the system is under normal load otherwise.

12. The method of claim 8, wherein the guaranteed processor time budget associated with each of the plurality of adaptive partitions is dynamic and can be reallocated while the system executes.

13. The method of claim 12, wherein the process thread to which the processor is allocated runs until the process thread is finished, is blocked, or the guaranteed processor time budget associated with adaptive partition with which the process thread is associated is exhausted.

14. The method of claim 13, wherein the process thread to which the processor is allocated further runs until the process thread is preempted and the processor is allocated to another process thread, of the plurality of process threads, that is in the ready state and that has a priority higher than the priority of the process thread to which the processor is currently allocated.

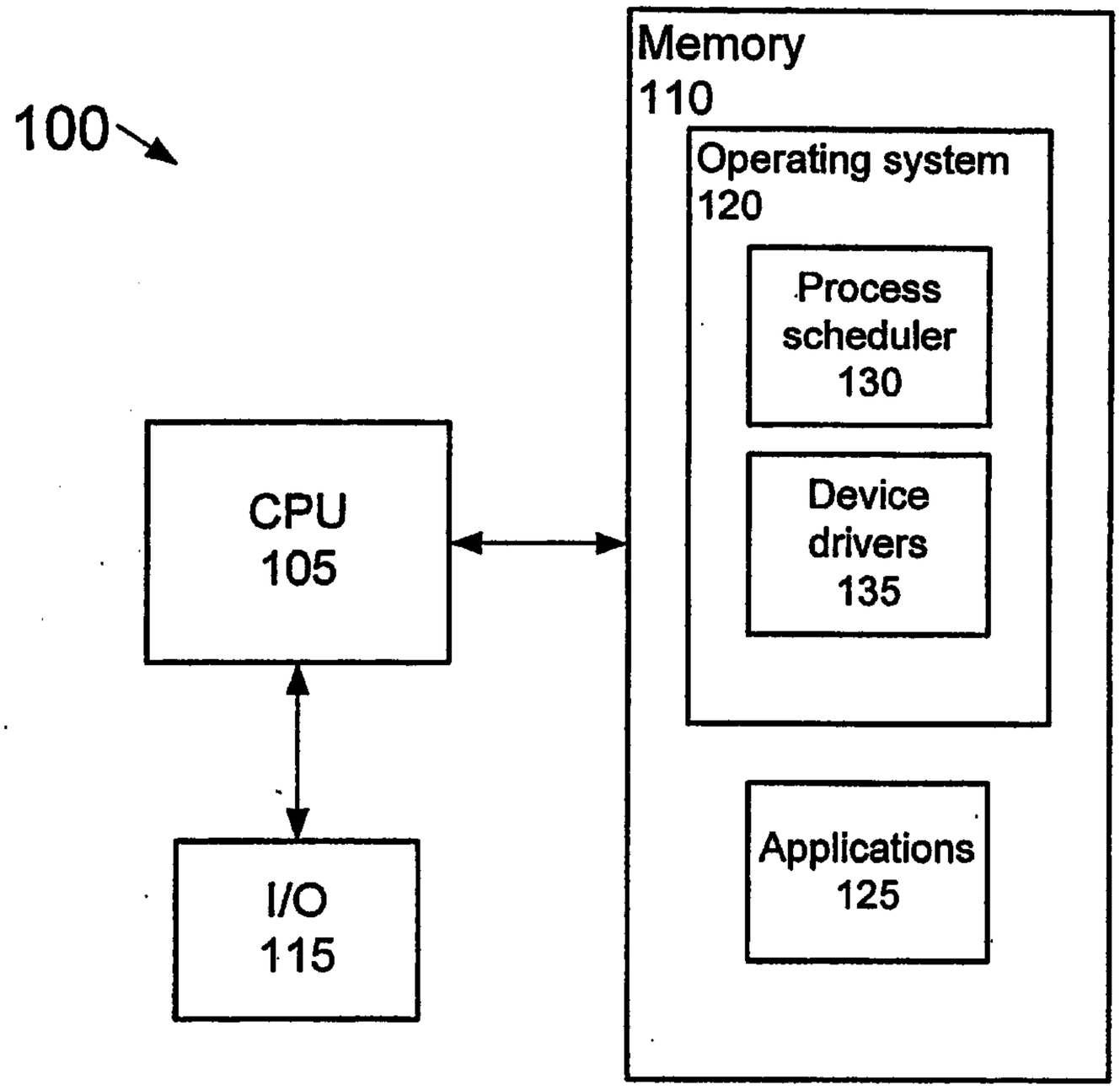

100
Memory
110
Operating system
120
Process scheduler
130
Device drivers
135
Applications
125
CPU
105
I/O
115

Figure 1

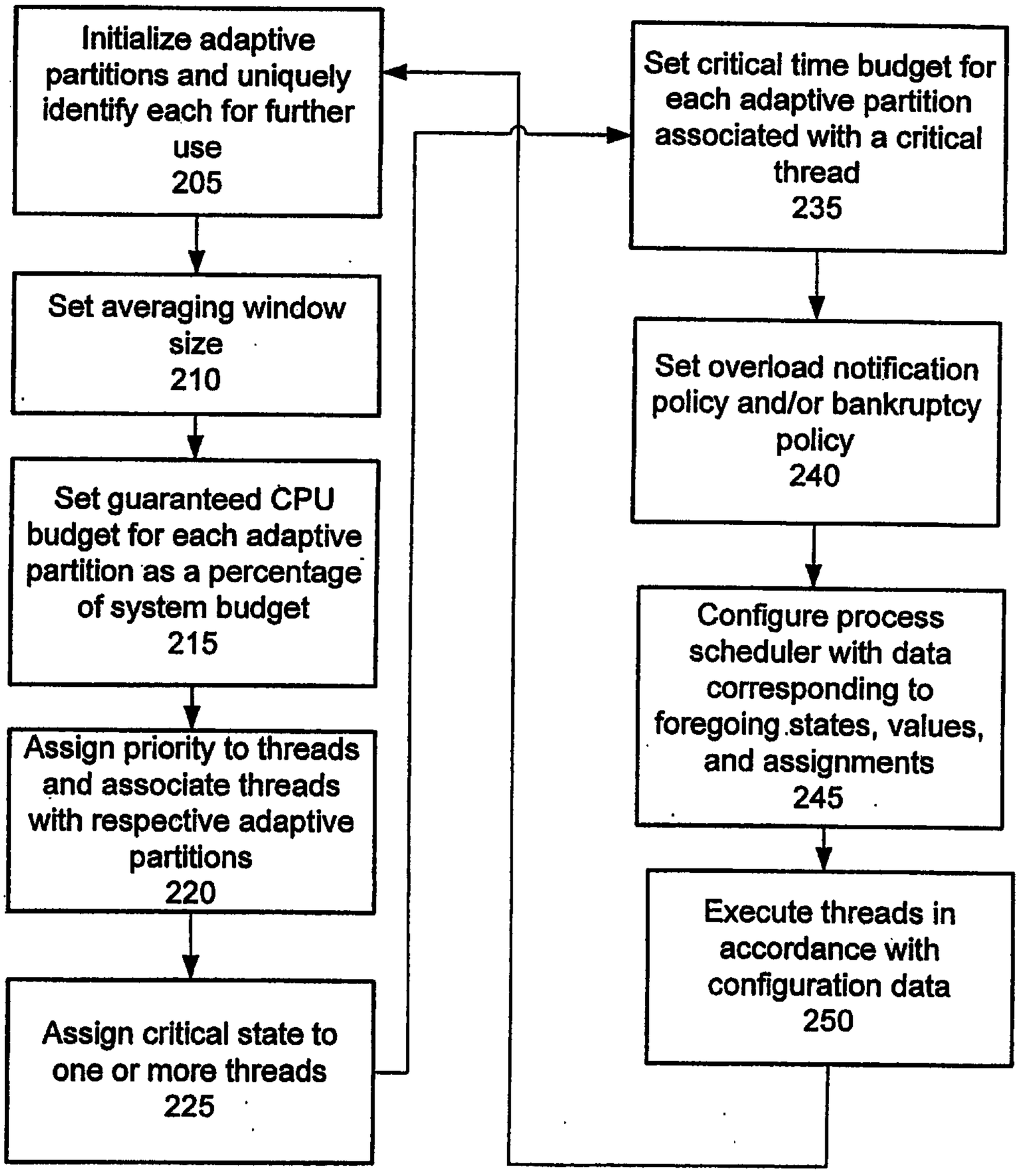

Initialize adaptive partitions and uniquely identify each for further use
205
Set averaging window size
210
Set guaranteed CPU budget for each adaptive partition as a percentage of system budget
215
Assign priority to threads and associate threads with respective adaptive partitions
220
Assign critical state to one or more threads
225
Set critical time budget for each adaptive partition associated with a critical thread
235
Set overload notification policy and/or bankruptcy policy
240
Configure process scheduler with data corresponding to foregoing states, values, and assignments
245
Execute threads in accordance with configuration data
250

Figure 2

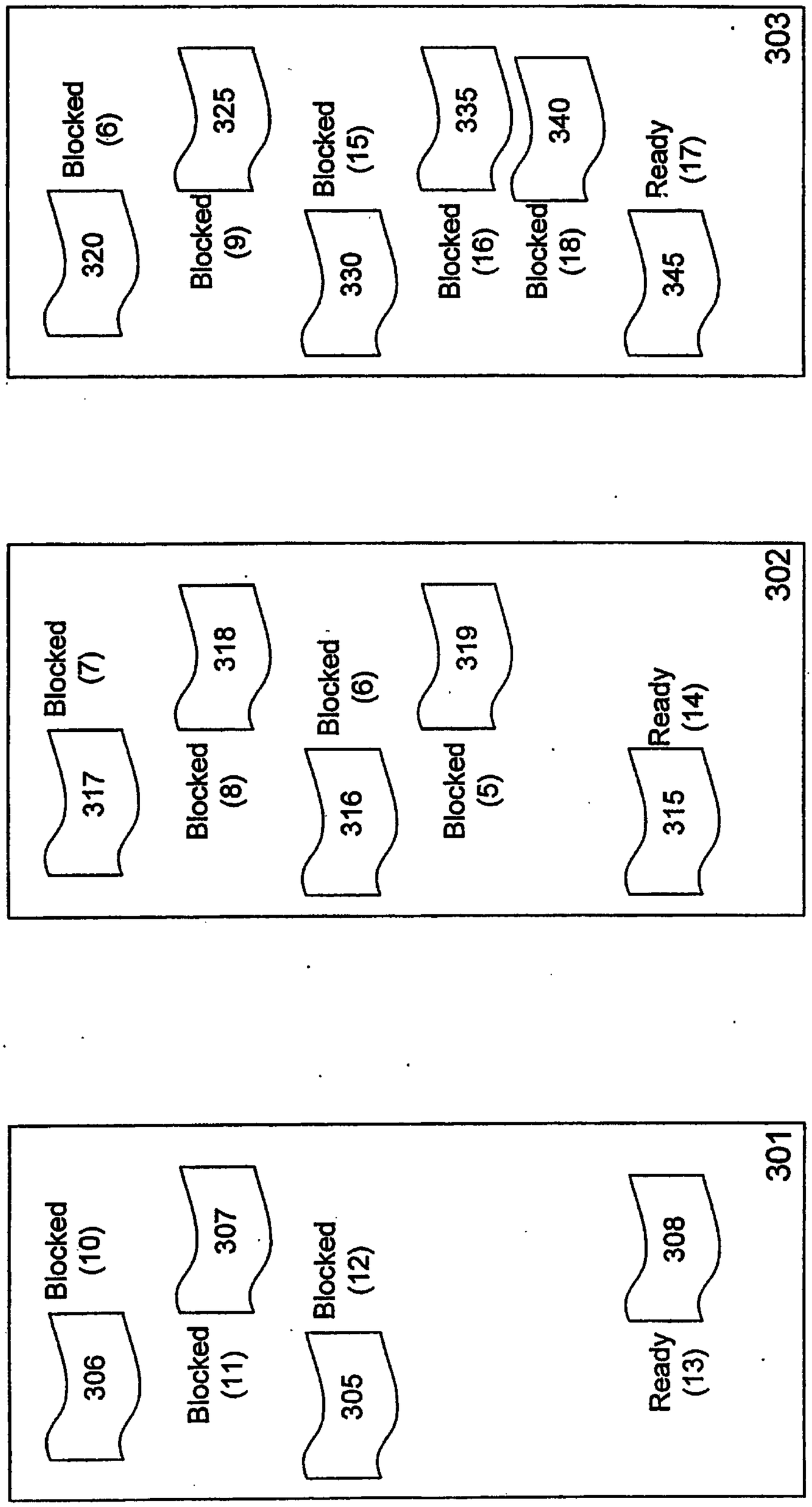

303
Blocked (6)
320
325
Blocked (9)
Blocked (15)
330
335
Blocked (16)
340
Blocked (18)
Ready (17)
345
302
Blocked (7)
317
318
Blocked (8)
Blocked (6)
316
319
Blocked (5)
Ready (14)
315
301
Blocked (10)
306
307
Blocked (11)
Blocked (12)
305
308
Ready (13)

Figure 3

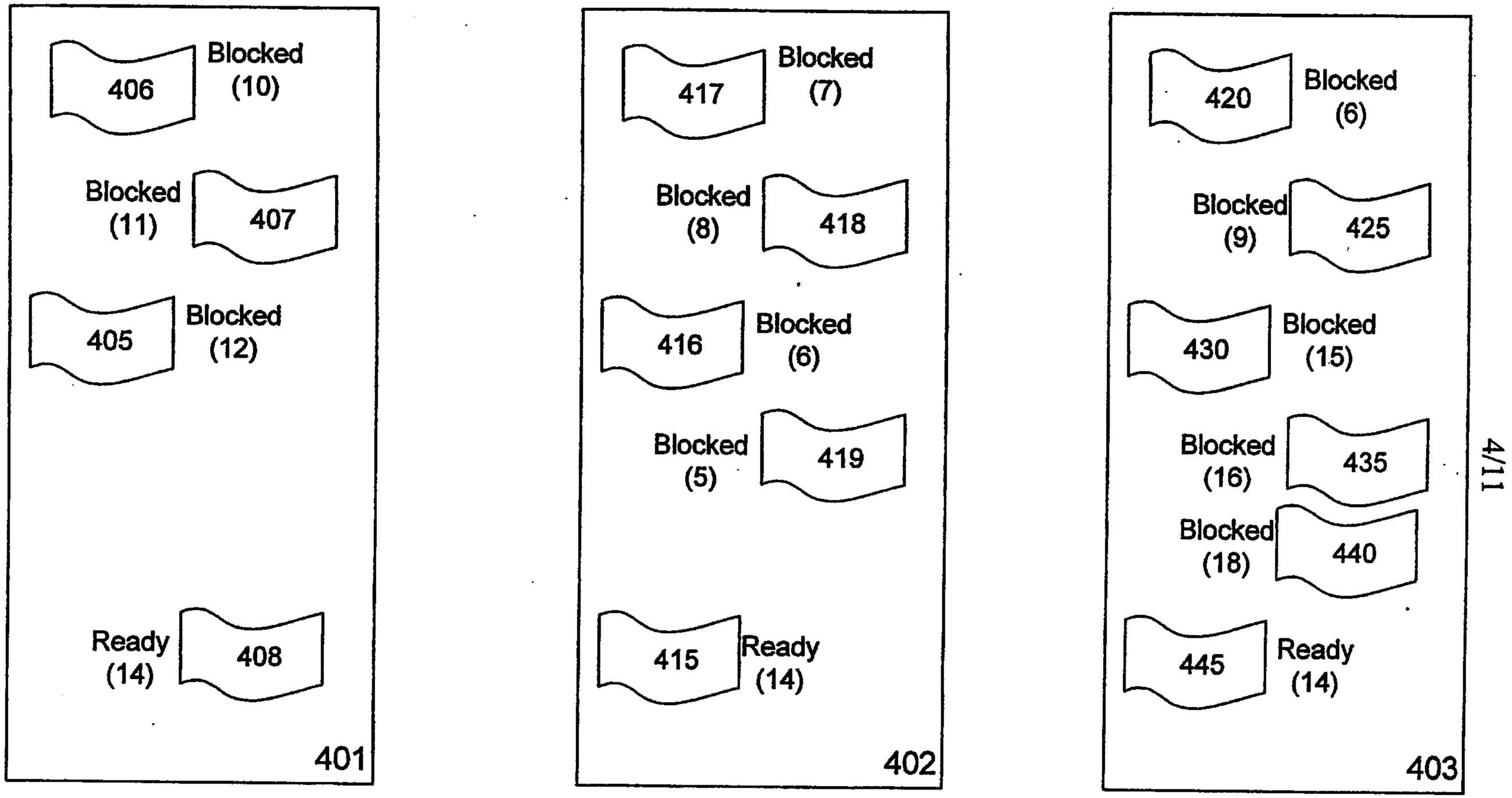


Figure 4

CA 02538503 2006-03-01

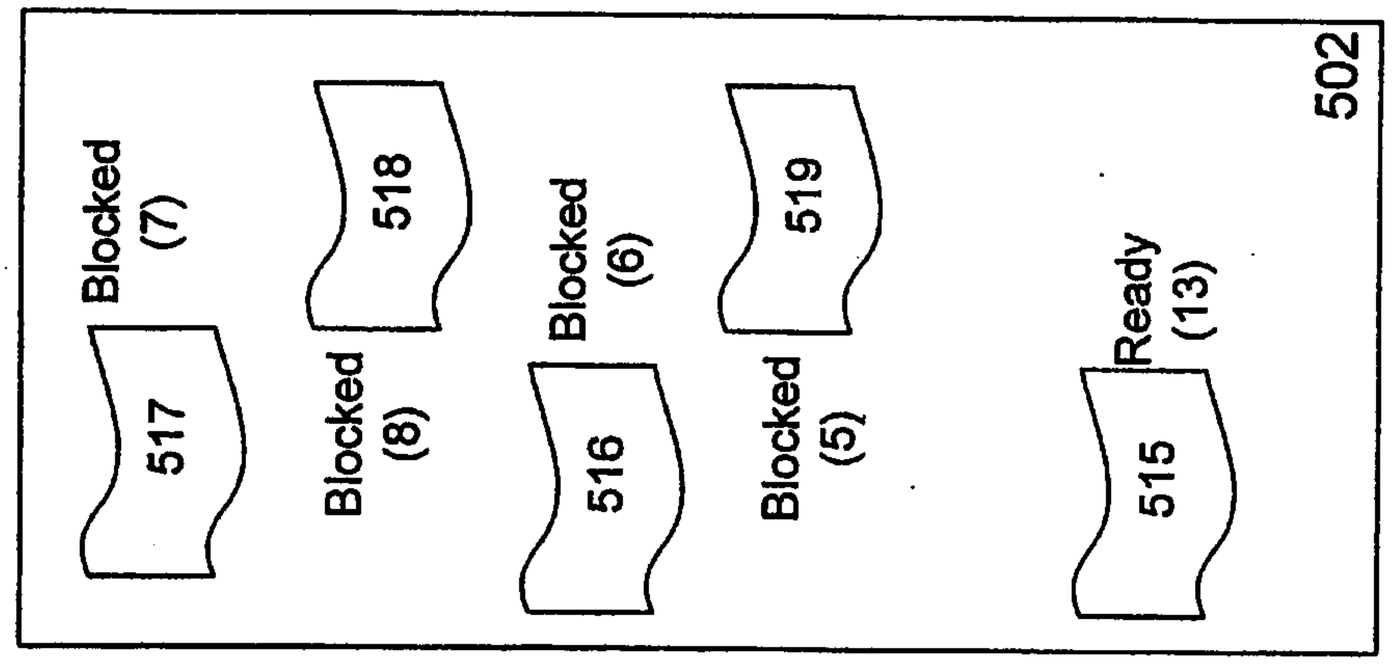

502
Blocked (7)
517
Blocked (8)
518
Blocked (6)
516
Blocked (5)
519
Ready (13)
515
501
Blocked (10)
506
Blocked (11)
507
Blocked Critical (12)
505
Ready Critical (14)
508

Figure 5

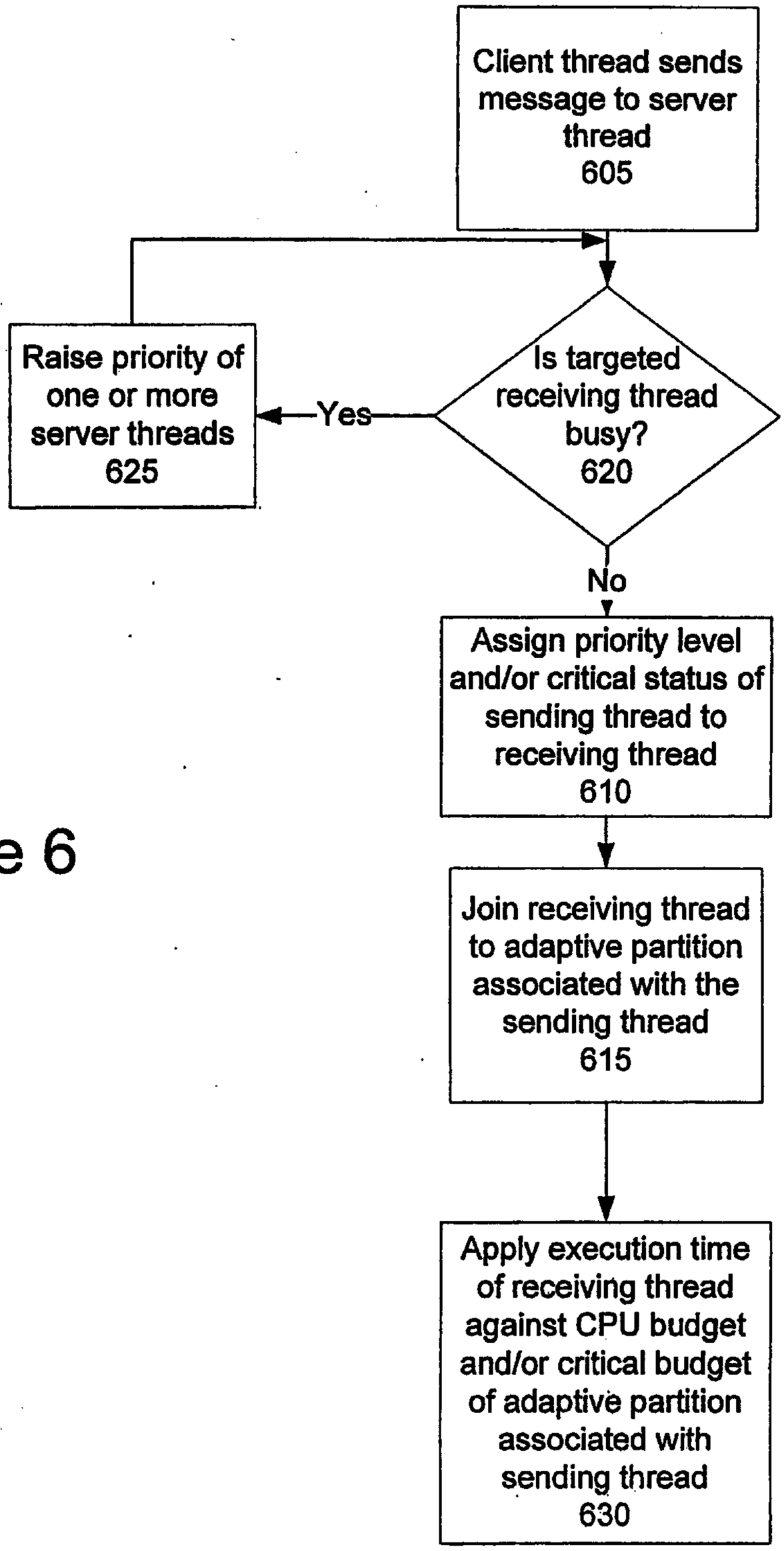

Client thread sends message to server thread 605
Is targeted receiving thread busy? 620
Yes
Raise priority of one or more server threads 625
No
Assign priority level and/or critical status of sending thread to receiving thread 610
Join receiving thread to adaptive partition associated with the sending thread 615
Apply execution time of receiving thread against CPU budget and/or critical budget of adaptive partition associated with sending thread 630

Figure 6

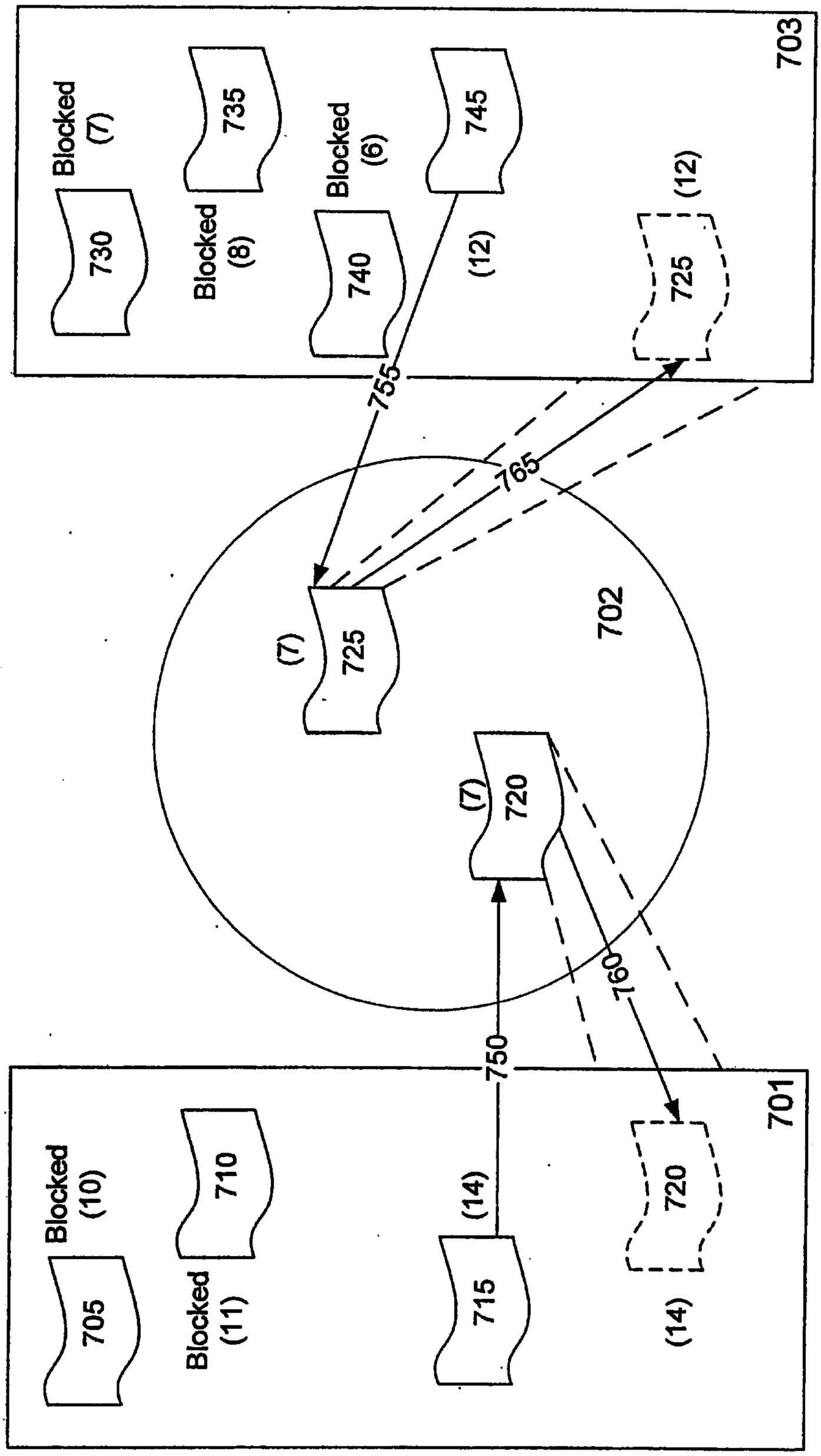
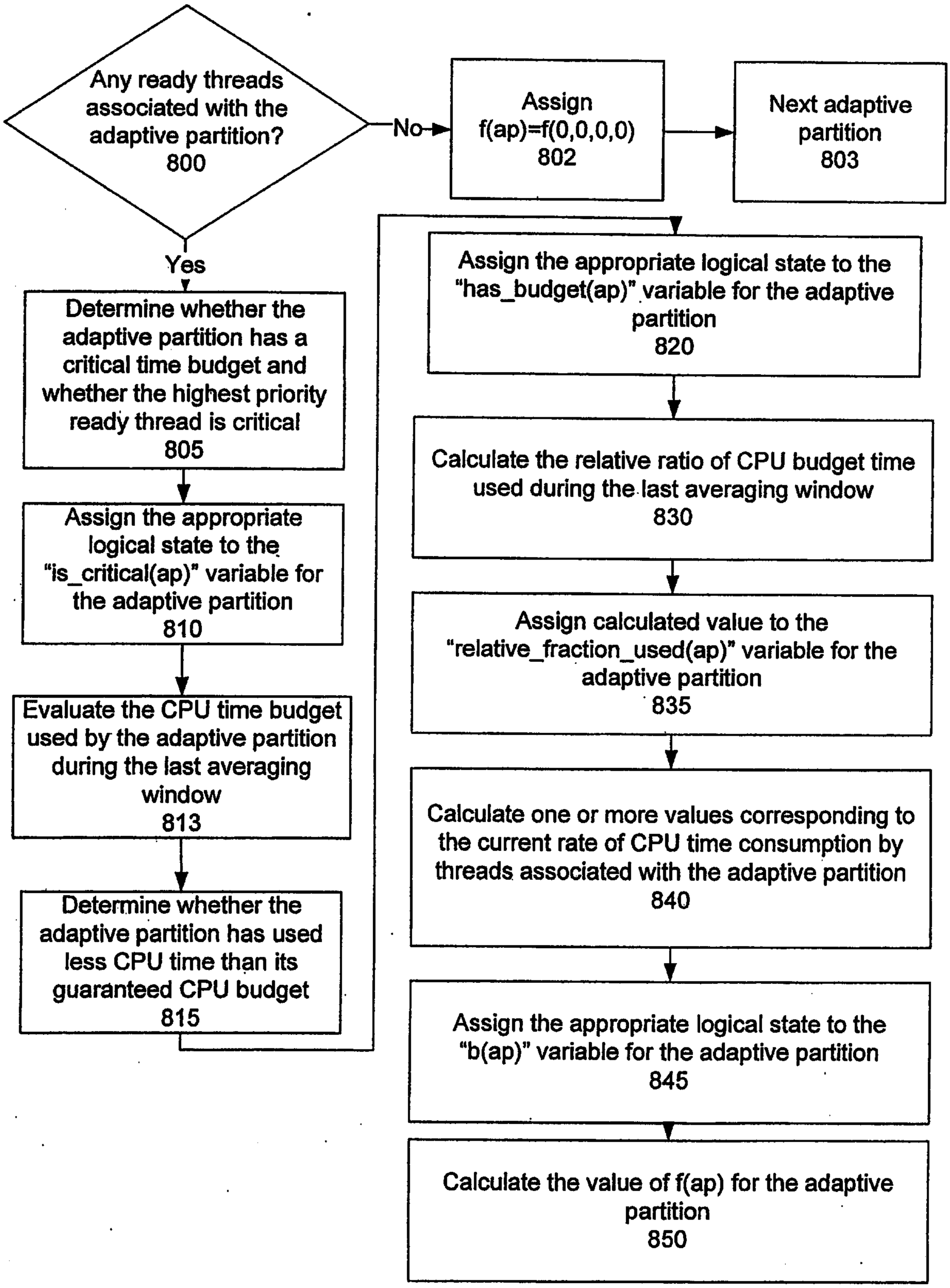

703
Blocked (7)
730
735
Blocked (8)
Blocked (6)
740
745
(12)
(12)
725
755
765
702
(7)
725
(7)
720
760
750
701
Blocked (10)
705
710
Blocked (11)
715
(14)
720
(14)

Figure 7


Any ready threads associated with the adaptive partition?
800
No
Assign f(ap)=f(0,0,0,0)
802
Next adaptive partition
803
Yes
Determine whether the adaptive partition has a critical time budget and whether the highest priority ready thread is critical
805
Assign the appropriate logical state to the "is_critical(ap)" variable for the adaptive partition
810
Evaluate the CPU time budget used by the adaptive partition during the last averaging window
813
Determine whether the adaptive partition has used less CPU time than its guaranteed CPU budget
815
Assign the appropriate logical state to the "has_budget(ap)" variable for the adaptive partition
820
Calculate the relative ratio of CPU budget time used during the last averaging window
830
Assign calculated value to the "relative_fraction_used(ap)" variable for the adaptive partition
835
Calculate one or more values corresponding to the current rate of CPU time consumption by threads associated with the adaptive partition
840
Assign the appropriate logical state to the "b(ap)" variable for the adaptive partition
845
Calculate the value of f(ap) for the adaptive partition
850

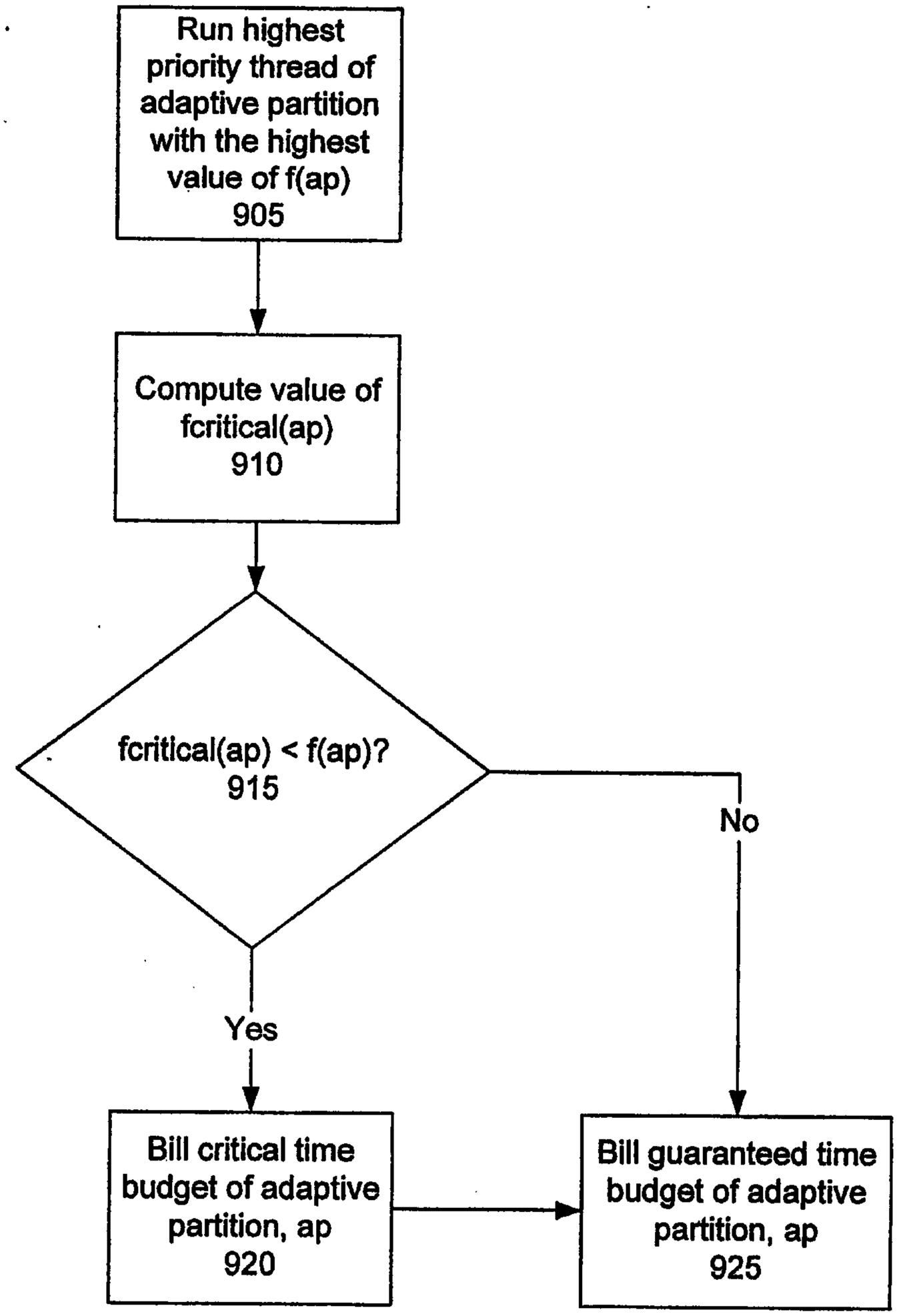

Run highest priority thread of adaptive partition with the highest value of f(ap)
905
Compute value of fcritical(ap)
910
fcritical(ap) < f(ap)?
915
No
Yes
Bill critical time budget of adaptive partition, ap
920
Bill guaranteed time budget of adaptive partition, ap
925

Figure 9

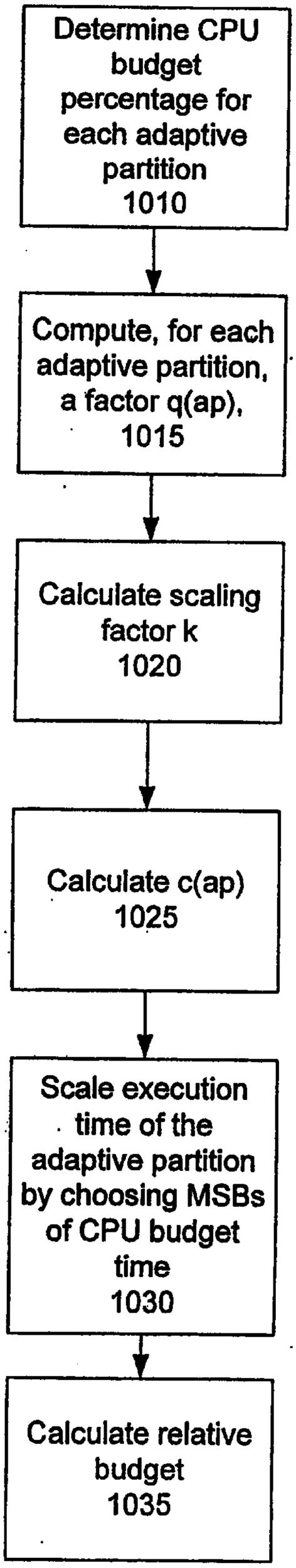

Determine CPU budget percentage for each adaptive partition 1010
Compute, for each adaptive partition, a factor q(ap), 1015
Calculate scaling factor k 1020
Calculate c(ap) 1025
Scale execution time of the adaptive partition by choosing MSBs of CPU budget time 1030
Calculate relative budget 1035

Figure 10

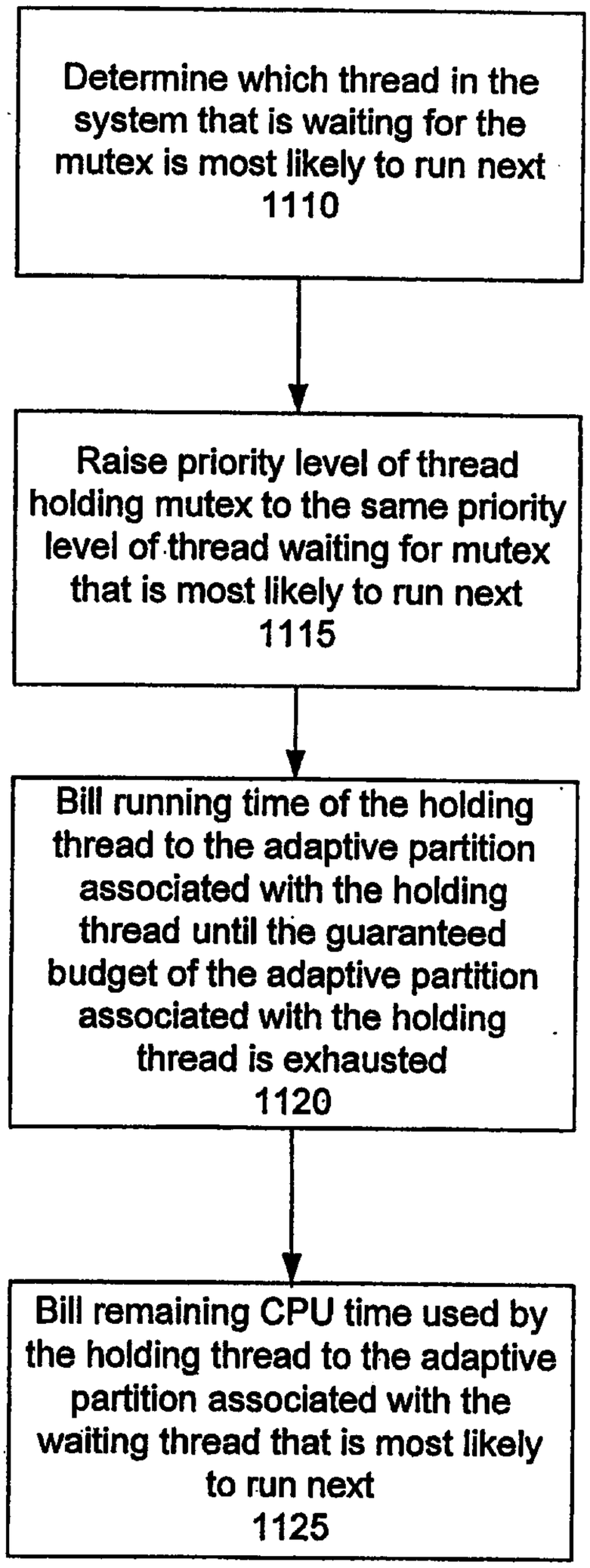

Determine which thread in the system that is waiting for the mutex is most likely to run next
1110
Raise priority level of thread holding mutex to the same priority level of thread waiting for mutex that is most likely to run next
1115
Bill running time of the holding thread to the adaptive partition associated with the holding thread until the guaranteed budget of the adaptive partition associated with the holding thread is exhausted
1120
Bill remaining CPU time used by the holding thread to the adaptive partition associated with the waiting thread that is most likely to run next
1125

Figure 11

Initialize adaptive partitions and uniquely identify each for further use
205

Set averaging window size
210

Set guaranteed CPU budget for each adaptive partition as a percentage of system budget
215

Assign priority to threads and associate threads with respective adaptive partitions
220

Assign critical state to one or more threads
225

Set critical time budget for each adaptive partition associated with a critical thread
235

Set overload notification policy and/or bankruptcy policy
240

Configure process scheduler with data corresponding to foregoing.states, values, and assignments
245

Execute threads in accordance with configuration data
250