PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/46

A1

(11) International Publication Number: WO 96/00941

(43) International Publication Date: 11 January 1996 (11.01.96)

(21) International Application Number: PCT/US95/08287

(22) International Filing Date: 30 June 1995 (30.06.95)

(30) Priority Data:
08/268,442 30 June 1994 (30.06.94) US

(71) Applicant: MICROSOFT CORPORATION [US/US]; One Microsoft Way, Redmond, WA 98052-6399 (US).

(72) Inventors: SCHMIDT, Michael, A.; 1646 - 204th Avenue N.E., Redmond, WA 98053 (US). THOMASON, Jonathan, G.; 7845 - 235th Place N.E., Redmond, WA 98053 (US). CUTSHALL, Scott, M.; 816 - 289th Avenue N.E., Carnation, WA 98014 (US).

(74) Agents: CANNING, Kevin, J. et al.; Seed and Berry, 6300 Columbia Center, 701 Fifth Avenue, Seattle, WA 98104-7092 (US).

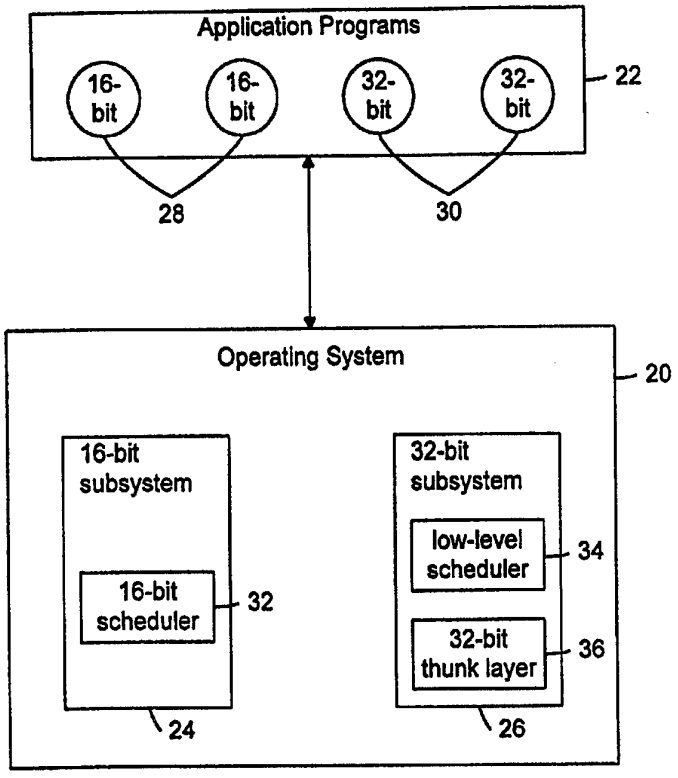
(81) Designated States: JP, European patent (AT, BE, CH, DE, DK, ES, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: METHOD AND SYSTEM FOR PROTECTING SHARED CODE AND DATA IN A MULTITASKING OPERATING SYSTEM

(57) Abstract

A method and system for protecting shared code and data, in particular, shared system code and data, in a multitasking operating system are provided. The operating system includes a cooperative subsystem and a preemptive subsystem. The cooperative subsystem includes shared system code and data. The method and system include a synchronization mechanism for controlling access to the shared system code and data by threads. Ownership of the synchronization mechanism must be requested and obtained before a cooperatively scheduled thread can execute in the cooperative subsystem. Additionally, ownership of the synchronization mechanism must be requested and obtained before a preemptively scheduled thread can execute the shared system code in the cooperative subsystem. If the synchronization mechanism is already owned, the requesting thread is blocked until ownership is released. Otherwise, the requesting thread is granted ownership. Since no other thread can obtain ownership of the synchronization mechanism while one thread owns the synchronization mechanism, the shared system code and data in the cooperative subsystem is protected.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AT	Austria	**GB**	United Kingdom	**MR**	Mauritania
AU	Australia	**GE**	Georgia	**MW**	Malawi
BB	Barbados	**GN**	Guinea	**NE**	Niger
BE	Belgium	**GR**	Greece	**NL**	Netherlands
BF	Burkina Faso	**HU**	Hungary	**NO**	Norway
BG	Bulgaria	**IE**	Ireland	**NZ**	New Zealand
BJ	Benin	**IT**	Italy	**PL**	Poland
BR	Brazil	**JP**	Japan	**PT**	Portugal
BY	Belarus	**KE**	Kenya	**RO**	Romania
CA	Canada	**KG**	Kyrgystan	**RU**	Russian Federation
CF	Central African Republic	**KP**	Democratic People's Republic of Korea	**SD**	Sudan
CG	Congo			**SE**	Sweden
CH	Switzerland	**KR**	Republic of Korea	**SI**	Slovenia
CI	Côte d'Ivoire	**KZ**	Kazakhstan	**SK**	Slovakia
CM	Cameroon	**LI**	Liechtenstein	**SN**	Senegal
CN	China	**LK**	Sri Lanka	**TD**	Chad
CS	Czechoslovakia	**LU**	Luxembourg	**TG**	Togo
CZ	Czech Republic	**LV**	Latvia	**TJ**	Tajikistan
DE	Germany	**MC**	Monaco	**TT**	Trinidad and Tobago
DK	Denmark	**MD**	Republic of Moldova	**UA**	Ukraine
ES	Spain	**MG**	Madagascar	**US**	United States of America
FI	Finland	**ML**	Mali	**UZ**	Uzbekistan
FR	France	**MN**	Mongolia	**VN**	Viet Nam
GA	Gabon				

METHOD AND SYSTEM FOR PROTECTING SHARED CODE AND DATA IN A MULTITASKING OPERATING SYSTEM

Field of the Invention

This invention relates generally to multitasking operating systems and, more particularly, to a method and system for protecting shared code and data in a multitasking operating system.

Background of the Invention

Many operating systems support multitasking. A task is a stand-alone application program or a subprogram that is run as an independent entity. Multitasking is the ability of a computer system to have more than one task in memory at a time and to switch which task the CPU is currently executing to improve the efficiency of the computer system's components. Multitasking operating systems provide mechanisms for protecting shared code and data. Shared code is code that can be executed by more than one task, and shared data is data that can be accessed by more than one task. The mechanisms provided by multitasking operating systems prevent problems associated with code sharing and data sharing. Code sharing and data sharing problems can arise if one task has modified shared code or the data that the shared code manipulates and another task begins or continues execution of the shared code after the shared code or its data has been modified. The task that has begun or continued execution of the shared code after modification of the code or its data can get unexpected results due to the modifications to the code or its data.

One type of multitasking operating system is a non-preemptive, or cooperative, multitasking operating system. A cooperative multitasking operating system requires cooperation between the tasks in order for the computer system to function properly. With such an

operating system, once a first task has been assigned control of the CPU, all other tasks are blocked (i.e., prevented from gaining control of the CPU) until the first task relinquishes control of the CPU. Thus, shared code and data are inherently protected in such an operating system. Specifically, since control of the CPU cannot be taken away from a task once it has been assigned, it is impossible for one task to have modified shared code or its data and another task to begin or continue execution of the shared code after the shared code or its data has been modified. The task that has been assigned control of the CPU will not relinquish control of the CPU until the task has completed execution of the shared code.

Another type of multitasking operating system is a preemptive multitasking operating system. Unlike a cooperative system, a preemptive multitasking operating system does not require cooperation between the tasks in order for the computer system to function properly. With such an operating system, the system assigns control of the CPU to a task and then takes back control from the task when a specified time expires or a specified event occurs. Thus, shared code and data are not inherently protected in such an operating system. For example, one task may have modified shared code or its data when the operating system takes back control of the CPU from the task. Subsequently, another task may be assigned control of the CPU and may begin or continue execution of the shared code after the shared code or its data has been modified. As a result, this situation can lead to the code sharing and data sharing problems discussed above.

Summary of the Invention

One aspect of the present invention provides a method and system for protecting shared code and data, in particular, shared system code and data, in a

multitasking operating system. The operating system includes a cooperative subsystem and a preemptive subsystem. The cooperative subsystem includes shared system code and data. The method and system include a synchronization mechanism for controlling access to the shared system code and data by threads. Ownership of the synchronization mechanism must be requested and obtained before a cooperatively scheduled thread can execute in the cooperative subsystem. Additionally, ownership of the synchronization mechanism must be requested and obtained before a preemptively scheduled thread can execute the shared system code in the cooperative subsystem. If the synchronization mechanism is already owned, the requesting thread is blocked until ownership is released. Otherwise, the requesting thread is granted ownership. Since no other thread can obtain ownership of the synchronization mechanism while one thread owns the synchronization mechanism, the shared system code and data in the cooperative subsystem are protected.

Brief Description of the Drawings

Figure 1 is a block diagram that illustrates the components of a computer system in which the preferred embodiment of the present invention operates;

Figure 2 is a block diagram that illustrates in more detail the components of the operating system of Figure 1;

Figure 3 is a high-level flow chart illustrating how a 16-bit thread requests, obtains, and releases ownership of a mutex in accordance with the preferred embodiment of the present invention; and

Figures 4A and 4B collectively are a high-level flow chart illustrating how a 32-bit thread requests, obtains, and releases ownership of the mutex in accordance with the preferred embodiment of the present invention.

Detailed Description of the Preferred Embodiment

The preferred embodiment of the present invention provides a method and system for protecting shared code and data, in particular, shared operating system code and data, in a multitasking operating system. The method and system include a synchronization mechanism that protects shared system code and data in the multitasking operating system. The operating system includes a cooperative subsystem and a preemptive subsystem. The cooperative subsystem includes the shared system code and data. As will be described in greater detail below, ownership of the synchronization mechanism must be requested and obtained before a cooperatively scheduled thread can execute in the cooperative subsystem. Additionally, ownership of the synchronization mechanism must be requested and obtained before a preemptively scheduled thread can execute the shared system code in the cooperative subsystem. The synchronization mechanism of the present invention provides compatibility in an operating system that includes both a cooperative subsystem and a preemptive subsystem by protecting the shared code and data in the cooperative subsystem from the preemptive capabilities of the preemptive subsystem.

A computer system 10 in which the preferred embodiment of the present invention operates is illustrated in Figure 1. The computer system 10 includes a central processing unit ("CPU") 12, a primary storage 14, a secondary storage 16, and input/output ("I/O") devices 18. An operating system 20 and application programs 22 are stored in the secondary storage 16 and are loaded into the primary storage 14 for execution by the CPU 12. A program, such as an application program 22, that has been loaded into the primary storage 14 and prepared for execution by the CPU 12 is called a process. A process includes the code, data, and other resources

that belong to a program. A path of execution in a process is called a thread. A thread includes a set of instructions, related CPU register values, and a stack. A process has at least one thread. In a multithreaded operating system, a process can have more than one thread. The thread is the entity that receives control of the CPU 12.

The components of the operating system 20 are illustrated in Figure 2. In the preferred embodiment of the present invention, the operating system 20 includes a cooperative multitasking subsystem 24 and a preemptive multitasking subsystem 26. Additionally, the cooperative multitasking subsystem 24 is a 16-bit system (and will be referred to as the "16-bit subsystem") and the preemptive multitasking subsystem 26 is a 32-bit system (and will be referred to as the "32-bit subsystem"). Thus, the application programs 22 include both cooperatively scheduled 16-bit applications 28 and preemptively scheduled 32-bit applications 30. The operating system 20 is an enhanced version of the "MICROSOFT WINDOWS," version 3.1, operating system (hereinafter referred to as "WINDOWS 3.1"), sold by Microsoft Corporation of Redmond, Washington. WINDOWS 3.1 is a cooperative multitasking operating system. The operating system 20 in the preferred embodiment of the present invention provides a preemptive multitasking subsystem as an enhancement to the cooperative multitasking subsystem provided by WINDOWS 3.1. Such an enhanced version provides compatibility between applications written for earlier cooperative systems and applications written for more recent preemptive systems. While the present invention is being described with reference to an enhanced version of WINDOWS 3.1, those skilled in the art will appreciate that other operating systems may be used to practice the present invention. The choice of an enhanced version of WINDOWS 3.1 is merely illustrative.

The 16-bit subsystem 24 includes a 16-bit application programming interface ("API") that enables the 16-bit applications 28 to request and carry out low-level services provided by the operating system 20. Similarly, the 32-bit subsystem 26 includes a 32-bit API that enables the 32-bit applications 30 to request and carry out low-level services provided by the operating system 20. Furthermore, the 16-bit subsystem includes a 16-bit scheduler 32 that is responsible for handling the cooperative scheduling of the 16-bit applications 28, and the 32-bit subsystem 26 includes a low-level scheduler 34 that is responsible for handling the preemptive scheduling of the 32-bit applications 30. Such a 16 bit scheduler is provided by the "MICROSOFT WINDOWS," version 3.1, operating system, and such a low-level scheduler is provided by the "MICROSOFT WINDOWS NT" operating system, sold by Microsoft Corporation of Redmond, Washington.

When a 16-bit application 28 calls a 16-bit API function, the call is handled directly by the 16-bit subsystem 24. When a 32-bit application 30 calls a 32-bit API function, however, the call is generally not handled directly by the 32-bit subsystem 26. Rather, if there is a 16-bit API function that has the same purpose as a 32-bit API function, the call to the 32-bit API function is typically converted into a call to the equivalent 16-bit API function and handled by the 16-bit subsystem 24. However, if there is no 16-bit API function that has the same purpose as the 32-bit API function, or if the 32-bit API function can be handled more efficiently by the 32-bit subsystem 26, the call to the 32-bit API function is handled directly by the 32-bit subsystem 26. The process of converting a call to a 32-bit API function into a call to an equivalent 16-bit API function and then converting the results of the 16-bit API function back into a 32-bit format is known as "thunking." This process occurs in a 32-bit thunk layer

36, which is code that performs the conversion to and from a 16-bit format. Such a thunk layer is provided by the "MICROSOFT WINDOWS NT" operating system.

Both the 16-bit API and the 32-bit API are implemented as a set of dynamic link libraries ("DLL's"). A DLL is a library module that contains executable code for performing various functions. The code for each function is stored within the DLL. When an application program contains a call to a function within a DLL, the DLL is dynamically linked to the application program at run time. Application programs use the functions in the DLL as if the functions were a part of the application program's code.

The DLL's that form the API's in the present invention are not reentrant. Code that is reentrant is not modifiable during run time and is written so that it can be shared by several threads. If one thread is executing reentrant code and another thread interrupts the execution of the first thread, the second thread can begin or continue execution of the same code without any code sharing or data sharing problems. Thus, more than one thread can be in the process of using a single DLL if the DLL is reentrant. The problem that occurs when a DLL is not reentrant is that the code is modifiable and the same code and data in the DLL are shared among all threads using the DLL. Consequently, if one thread is executing code in a non-reentrant DLL and another thread interrupts the execution of the first thread and attempts to begin or continue execution of the same code in the DLL, there may be code sharing or data sharing problems. Since the DLL's that form the 16-bit API in the present invention are not reentrant and since the operating system 20 includes both a cooperative 16-bit subsystem 24 and a preemptive 32-bit subsystem 26, a mechanism is needed to prevent a second thread from executing a 16-bit

API function before a first thread has completed execution of another 16-bit API function.

The method and system provided by the present invention for protecting the shared code and data in the 16-bit API include a synchronization mechanism in the form of a mutex. Those skilled in the art will appreciate that the present invention may also employ other synchronization mechanisms, such as critical sections or semaphores. A mutex is a data structure that can be used to prevent simultaneous use of a shared resource. In this case, the shared resource is the code and data in the 16-bit API. When a thread wants to use the shared resource, the thread requests ownership of the mutex. If the mutex is already owned by another thread, the requesting thread is blocked until ownership is released. Otherwise, the requesting thread is granted ownership. In the preferred embodiment of the present invention, the time and manner in which threads request, obtain, and release ownership of the mutex differs for 16-bit threads and 32-bit threads.

For 16-bit threads, ownership of the mutex must be requested and obtained before the 16-bit thread calls into its application code. Figure 3 illustrates the steps that must be performed for a 16-bit thread to request, obtain, and release ownership of the mutex. Initially, when a 16-bit application 28 is started, a 16-bit thread is created and linked into the 16-bit scheduler 32 (step 38). Before the 16-bit thread calls into its application code, the 16-bit thread requests ownership of the mutex inside the 16-bit scheduler 32 (step 40). If the mutex is already owned (step 42), the 16-bit thread is blocked until ownership is released and the thread has priority to execute (step 44). Otherwise, the 16-bit thread is granted ownership (step 46) and calls into its application code and executes (step 48). When the 16-bit thread relinquishes control of the CPU 12

to the 16-bit scheduler 32, the 16-bit thread releases ownership of the mutex inside the 16-bit scheduler 32 (step 50). While the 16-bit thread owns the mutex, the 16-bit thread can make multiple calls to functions within the 16-bit API. Since no other thread can obtain ownership of the mutex while the 16-bit thread owns the lock, the code and data in the 16-bit API are protected.

For 32-bit threads, ownership of the mutex must be requested and obtained before a call to a 16-bit API function (that has been thunked from a call to an equivalent 32-bit API function) begins execution. Figures 4A and 4B collectively illustrate the steps that must be performed for a 32-bit thread to request, obtain, and release ownership of the mutex. Initially, when a 32-bit application 30 is started, a 32-bit thread is created and linked into the low-level scheduler 34 (step 52). When the 32-bit thread is assigned control of the CPU 12, the 32-bit thread calls into its application code and executes (step 54). During execution, when the 32-bit thread calls a 32-bit API function that must be thunked (i.e., one that must be converted into a call to an equivalent 16-bit API function) (step 56), the 32-bit API function transfers control to the 32-bit thunk layer 36 (step 58). The 32-bit thunk layer 36 converts the call to the 32-bit API function into a call to the equivalent 16-bit API function (step 60) and the 32-bit thread requests ownership of the mutex inside the 32-bit thunk layer 36 (step 62). If the mutex is already owned (step 64), the 32-bit thread is blocked until ownership is released and the thread has priority to execute (step 66). Otherwise, the 32-bit thread is granted ownership (step 68) and calls the equivalent 16-bit API function (step 70). The 16-bit API function handles the call and returns the results to the 32-bit thunk layer 36 (step 72) and the 32-bit thread releases ownership of the mutex inside the 32-bit thunk layer 36 (step 74). Lastly, the

32-bit thunk layer 36 converts the results from the 16-bit API function back into a 32-bit format and returns control to the 32-bit thread (step 76). Again, since no other thread can obtain ownership of the mutex while the 32-bit thread owns the lock, the code and data in the 16-bit API are protected.

In the discussion above relating to 32-bit threads, ownership of the mutex is requested after the call to the 32-bit API function has been converted into a call to the equivalent 16-bit API function and is released before the results from the 16-bit API function are converted back into a 32-bit format. Those skilled in the art will appreciate that ownership of the mutex may also be requested before the call to the 32-bit API function has been converted into a call to the equivalent 16-bit API function and be released after the results from the 16-bit API function are converted back into a 32-bit format.

One of ordinary skill in the art will now appreciate that the present invention provides a method and system for protecting shared code and data, in particular, shared system code and data, in a multitasking operating system. The method and system include a synchronization mechanism that protects shared system code and data in the multitasking operating system. The operating system includes a cooperative subsystem and a preemptive subsystem. The cooperative subsystem includes the shared system code and data. Ownership of the synchronization mechanism must be requested and obtained before a cooperatively scheduled thread can execute in the cooperative subsystem. Additionally, ownership of the synchronization mechanism must be requested and obtained before a preemptively scheduled thread can execute the shared system code in the cooperative subsystem. The synchronization mechanism of the present invention provides compatibility in an

operating system that includes both a cooperative subsystem and a preemptive subsystem by protecting the shared code and data in the cooperative subsystem from the preemptive capabilities of the preemptive subsystem. Although the present invention has been shown and described with reference to a preferred embodiment, equivalent alterations and modifications will occur to those skilled in the art upon reading and understanding this specification. The present invention includes all such equivalent alterations and modifications and is limited only by the scope of the following claims.

What is claimed is:

1. In a computer system, a method for protecting shared system code and data in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the method comprising the steps of:

providing a synchronization mechanism for controlling access to the shared system code and data by threads;

providing a cooperatively scheduled thread to execute in the cooperative subsystem; and

requesting ownership of the synchronization mechanism before the cooperatively scheduled thread begins execution.

2. In a computer system, a method for protecting shared system code and data in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the method comprising the steps of:

providing a synchronization mechanism for controlling access to the shared system code and data by threads;

providing a cooperatively scheduled thread to execute in the cooperative subsystem;

requesting ownership of the synchronization mechanism before the cooperatively scheduled thread begins execution;

granting ownership of the synchronization mechanism to the cooperatively scheduled thread if the synchronization mechanism is not already owned;

allowing the cooperatively scheduled thread to execute if ownership of the synchronization mechanism was obtained; and

blocking the cooperatively scheduled thread from executing if ownership of the synchronization mechanism was not obtained.

3. The method of claim 2, further including the step of releasing ownership of the synchronization mechanism after the cooperatively scheduled thread has executed if ownership of the synchronization mechanism was obtained.

4. The method of claim 2, wherein the step of providing a synchronization mechanism comprises the step of providing a mutex for controlling access to the shared system code and data by threads.

5. The method of claim 2, wherein the shared system code and data include an application programming interface that enables threads to request services provided by the operating system, and wherein the application programming interface is implemented as a set of dynamic link libraries.

6. In a computer system, a method in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the method comprising the steps of:

providing shared system code and data in the cooperative subsystem, the shared system code and data including an application programming interface that enables threads to request services provided by the operating system, the application programming interface being implemented as a set of dynamic link libraries, the

dynamic link libraries including code for performing the services provided by the operating system;

providing a synchronization mechanism in the form of a mutex for controlling access to the dynamic link libraries by threads;

providing a cooperatively scheduled thread to execute in the cooperative subsystem;

requesting ownership of the mutex before the cooperatively scheduled thread begins execution;

granting ownership of the mutex to the cooperatively scheduled thread if the mutex is not already owned;

allowing the cooperatively scheduled thread to execute if ownership of the mutex was obtained; and

blocking the cooperatively scheduled thread from executing if ownership of the mutex was not obtained.

7. In a computer system, a method for protecting shared system code and data in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the method comprising the steps of:

providing a synchronization mechanism for controlling access to the shared system code and data by threads;

providing a preemptively scheduled thread to execute the shared system code in the cooperative subsystem; and

requesting ownership of the synchronization mechanism before the preemptively scheduled thread begins execution of the shared system code.

8. In a computer system, a method for protecting shared system code and data in a multitasking

operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the method comprising the steps of:

providing a synchronization mechanism for controlling access to the shared system code and data by threads;

providing a preemptively scheduled thread to execute the shared system code in the cooperative subsystem;

requesting ownership of the synchronization mechanism before the preemptively scheduled thread begins execution of the shared system code;

granting ownership of the synchronization mechanism to the preemptively scheduled thread if the synchronization mechanism is not already owned;

allowing the preemptively scheduled thread to execute the shared system code if ownership of the synchronization mechanism was obtained;

blocking the preemptively scheduled thread from executing the shared system code if ownership of the synchronization mechanism was not obtained.

9. The method of claim 8, further including the step of releasing ownership of the synchronization mechanism after the preemptively scheduled thread has executed the shared system code if ownership of the synchronization mechanism was obtained.

10. The method of claim 8, wherein the step of providing a synchronization mechanism comprises the step of providing a mutex for controlling access to the shared system code and data by threads.

11. The method of claim 8, wherein the shared system code and data include an application programming

interface that enables threads to request services provided by the operating system, and wherein the application programming interface is implemented as a set of dynamic link libraries.

12. In a computer system, a method in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the method comprising the steps of:

providing shared system code and data in the cooperative subsystem, the shared system code and data including an application programming interface that enables threads to request services provided by the operating system, the application programming interface being implemented as a set of dynamic link libraries, the dynamic link libraries including code for performing the services provided by the operating system;

providing a synchronization mechanism in the form of a mutex for controlling access to the dynamic link libraries by threads;

providing a preemptively scheduled thread to execute the code in the dynamic link libraries;

requesting ownership of the mutex before the preemptively scheduled thread begins execution of the code in the dynamic link libraries;

granting ownership of the mutex to the preemptively scheduled thread if the mutex is not already owned;

allowing the preemptively scheduled thread to execute the code in the dynamic link libraries if ownership of the mutex was obtained; and

blocking the preemptively scheduled thread from executing the code in the dynamic link libraries if ownership of the mutex was not obtained.

13. In a computer system, a method for protecting shared system code and data in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the method comprising the steps of:

providing a synchronization mechanism for controlling access to the shared system code and data by threads;

providing a cooperatively scheduled thread to execute in the cooperative subsystem;

providing a preemptively scheduled thread to execute the shared system code in the cooperative subsystem;

requesting ownership of the synchronization mechanism before the cooperatively scheduled thread begins execution;

granting ownership of the synchronization mechanism to the cooperatively scheduled thread if the synchronization mechanism is not already owned;

allowing the cooperatively scheduled thread to execute if ownership of the synchronization mechanism was obtained by the cooperatively scheduled thread;

blocking the cooperatively scheduled thread from executing if ownership of the synchronization mechanism was not obtained by the cooperatively scheduled thread;

requesting ownership of the synchronization mechanism before the preemptively scheduled thread begins execution of the shared system code;

granting ownership of the synchronization mechanism to the preemptively scheduled thread if the synchronization mechanism is not already owned;

allowing the preemptively scheduled thread to execute the shared system code if ownership of the

synchronization mechanism was obtained by the preemptively scheduled thread; and

blocking the preemptively scheduled thread from executing the shared system code if ownership of the synchronization mechanism was not obtained by the preemptively scheduled thread.

14. The method of claim 13, further including the steps of:

releasing ownership of the synchronization mechanism after the cooperatively scheduled thread has executed if ownership of the synchronization mechanism was obtained by the cooperatively scheduled thread; and

releasing ownership of the synchronization mechanism after the preemptively scheduled thread has executed the shared system code if ownership of the synchronization mechanism was obtained by the preemptively scheduled thread.

15. The method of claim 13, wherein the step of providing a synchronization mechanism comprises the step of providing a mutex for controlling access to the shared system code and data by threads.

16. The method of claim 13, wherein the shared system code and data include an application programming interface that enables threads to request services provided by the operating system, and wherein the application programming interface is implemented as a set of dynamic link libraries.

17. In a computer system, a method in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the method comprising the steps of:

providing shared system code and data in the cooperative subsystem, the shared system code and data including an application programming interface that enables threads to request services provided by the operating system, the application programming interface being implemented as a set of dynamic link libraries, the dynamic link libraries including code for performing the services provided by the operating system;

providing a synchronization mechanism in the form of a mutex for controlling access to the dynamic link libraries by threads;

providing a cooperatively scheduled thread to execute in the cooperative subsystem;

providing a preemptively scheduled thread to execute the code in the dynamic link libraries;

requesting ownership of the mutex before the cooperatively scheduled thread begins execution;

granting ownership of the mutex to the cooperatively scheduled thread if the mutex is not already owned;

allowing the cooperatively scheduled thread to execute if ownership of the mutex was obtained by the cooperatively scheduled thread;

blocking the cooperatively scheduled thread from executing if ownership of the mutex was not obtained by the cooperatively scheduled thread;

requesting ownership of the mutex before the preemptively scheduled thread begins execution of the code in the dynamic link libraries;

granting ownership of the mutex to the preemptively scheduled thread if the mutex is not already owned;

allowing the preemptively scheduled thread to execute the code in the dynamic link libraries if ownership of the mutex was obtained by the preemptively scheduled thread; and

blocking the preemptively scheduled thread from executing the code in the dynamic link libraries if ownership of the mutex was not obtained by the preemptively scheduled thread.

18. In a computer system, a system for protecting shared system code and data in a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the cooperative subsystem including the shared system code and data, the system comprising:

a synchronization mechanism for controlling access to the shared system code and data by threads;

a cooperatively scheduled thread to execute in the cooperative subsystem;

a preemptively scheduled thread to execute the shared system code in the cooperative subsystem;

a cooperative scheduler for requesting and obtaining ownership of the synchronization mechanism before allowing the cooperatively scheduled thread to begin execution and for releasing ownership of the synchronization mechanism after the cooperatively scheduled thread has executed; and

a thunk layer for requesting and obtaining ownership of the synchronization mechanism before allowing the preemptively scheduled thread to begin execution of the shared system code and for releasing ownership of the synchronization mechanism after the preemptively scheduled thread has executed the shared system code.

19. The system of claim 18, wherein the synchronization mechanism is a mutex.

20. The system of claim 18, wherein the shared system code and data include an application programming interface that enables threads to request services

provided by the operating system, and wherein the application programming interface is implemented a a set of dynamic link libraries.

21. In a computer system, a multitasking operating system having a cooperative subsystem and a preemptive subsystem, the operating system comprising:

shared system code and data in the cooperative subsystem, the shared system code and data including an application programming interface that enables threads to request services provided by the operating system, the application programming interface being implemented as a set of dynamic link libraries, the dynamic link libraries including code for performing the services provided by the operating system;

a synchronization mechanism in the form of a mutex for controlling access to the dynamic link libraries by threads executing the code in the dynamic link libraries;

a cooperatively scheduled thread to execute in the cooperative subsystem;

a preemptively scheduled thread to execute the code in the dynamic link libraries;

a cooperative scheduler for requesting and obtaining ownership of the mutex before allowing the cooperatively scheduled thread to begin execution; and

a thunk layer for requesting and obtaining ownership of the mutex before allowing the preemptively scheduled thread to begin execution of the code in the dynamic link libraries.

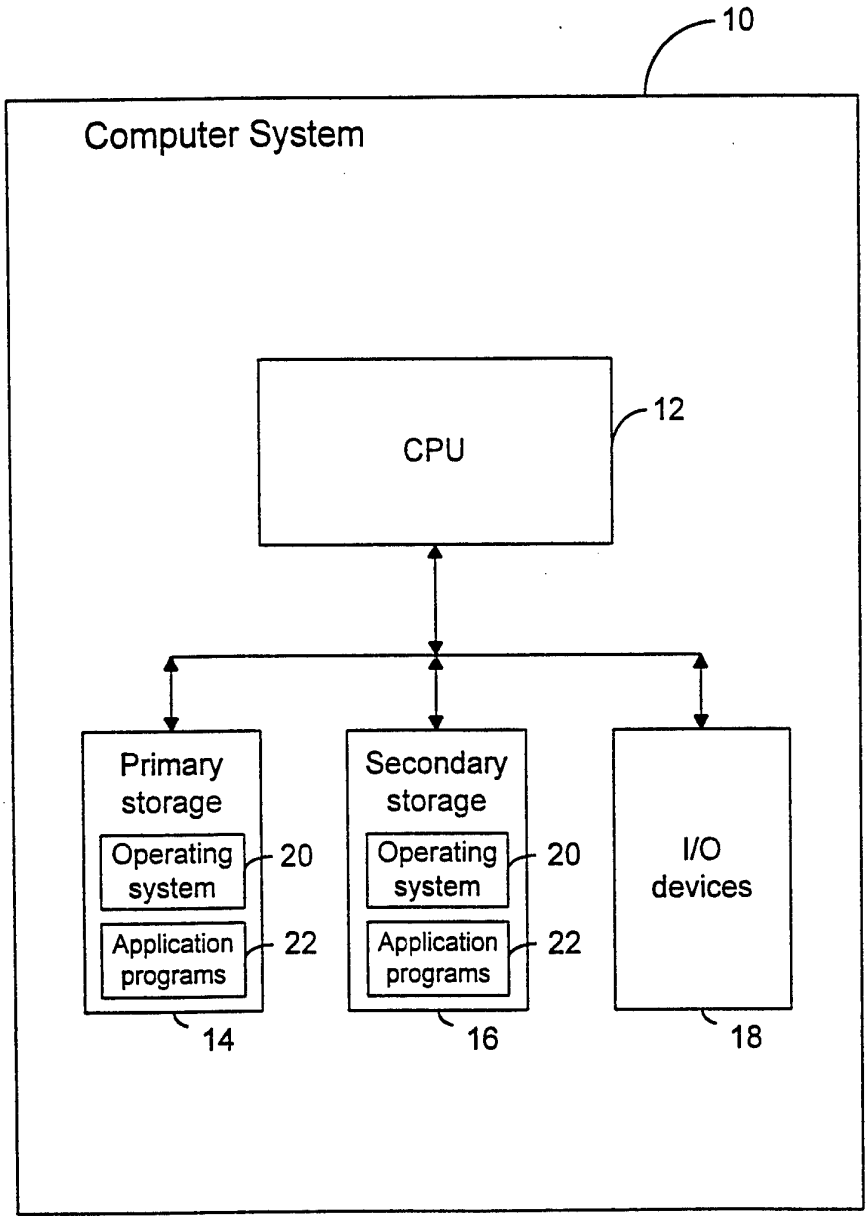

10
Computer System
CPU
12
Primary storage
Operating system
20
Application programs
22
14
Secondary storage
Operating system
20
Application programs
22
16
I/O devices
18

FIG. 1

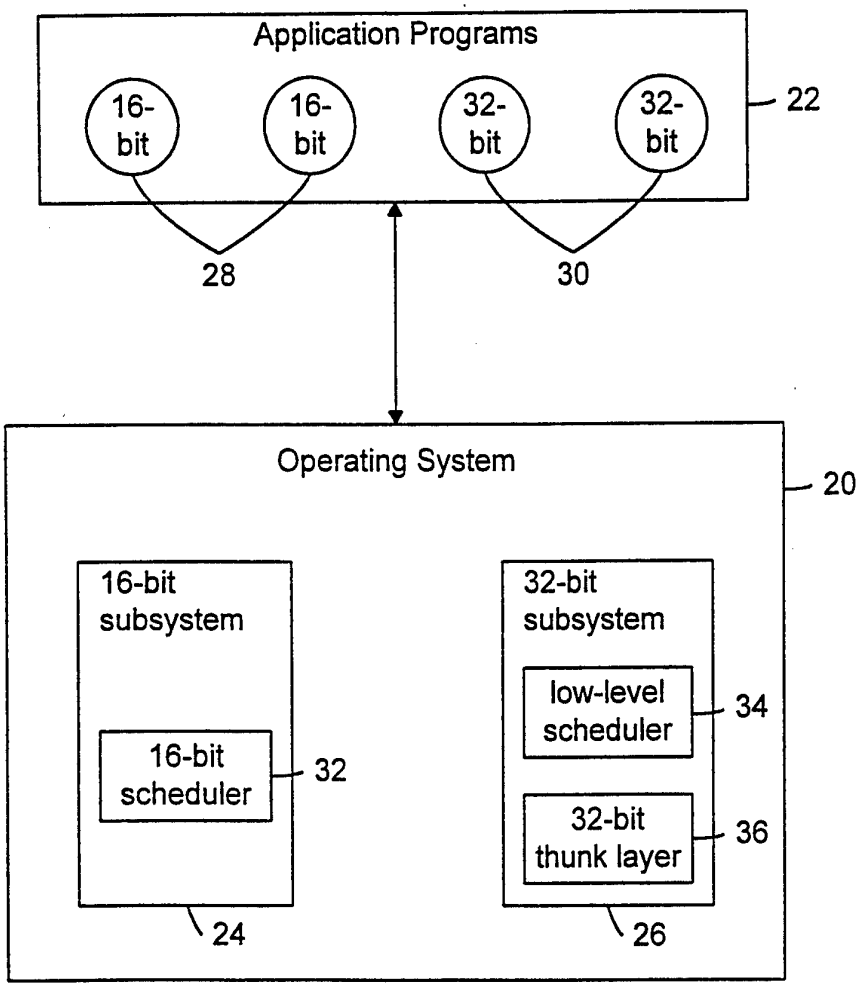

Application Programs
16-bit
16-bit
32-bit
32-bit
22
28
30
Operating System
20
16-bit subsystem
16-bit scheduler
32
24
32-bit subsystem
low-level scheduler
34
32-bit thunk layer
36
26

FIG. 2

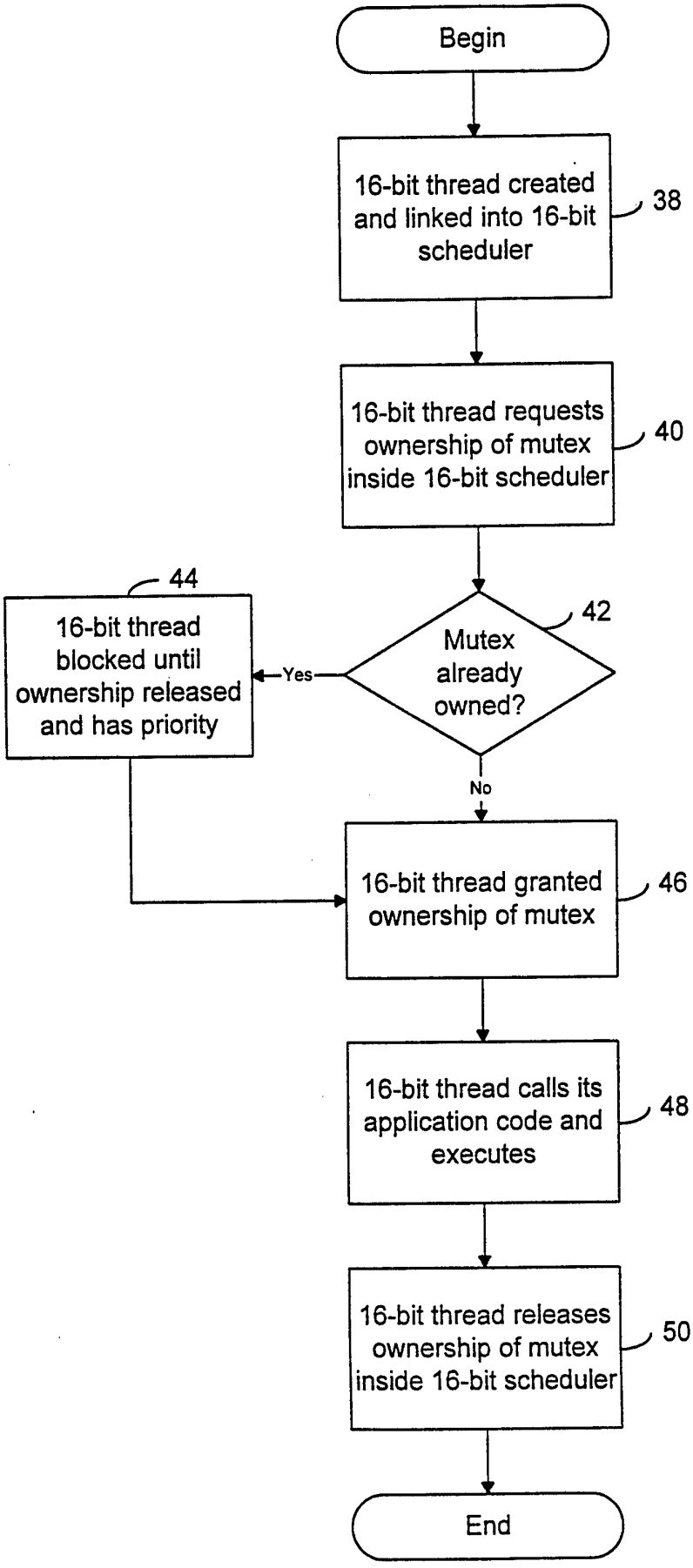

Begin
16-bit thread created and linked into 16-bit scheduler
38
16-bit thread requests ownership of mutex inside 16-bit scheduler
40
Mutex already owned?
42
Yes
16-bit thread blocked until ownership released and has priority
44
No
16-bit thread granted ownership of mutex
46
16-bit thread calls its application code and executes
48
16-bit thread releases ownership of mutex inside 16-bit scheduler
50
End

FIG. 3

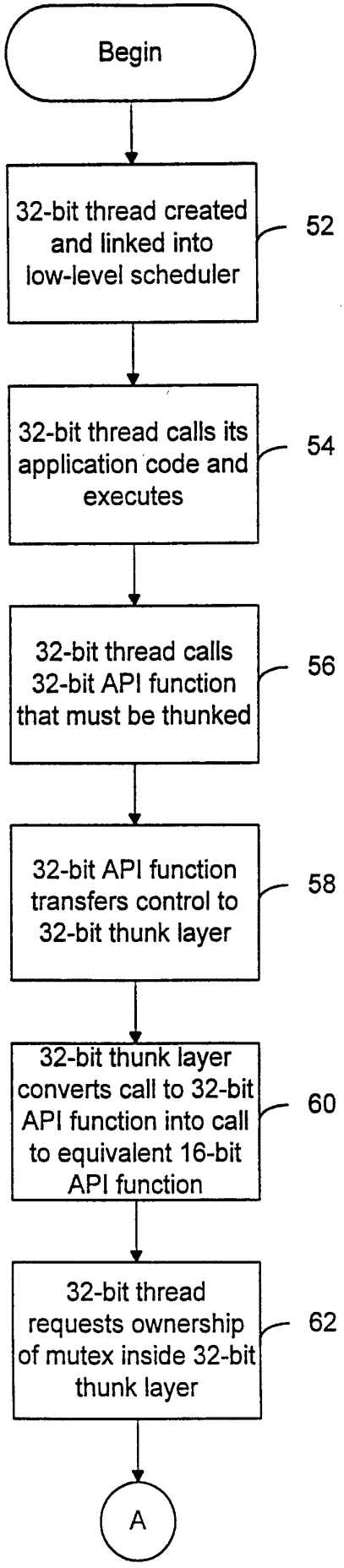

Begin
32-bit thread created and linked into low-level scheduler
52
32-bit thread calls its application code and executes
54
32-bit thread calls 32-bit API function that must be thunked
56
32-bit API function transfers control to 32-bit thunk layer
58
32-bit thunk layer converts call to 32-bit API function into call to equivalent 16-bit API function
60
32-bit thread requests ownership of mutex inside 32-bit thunk layer
62
A

FIG. 4A

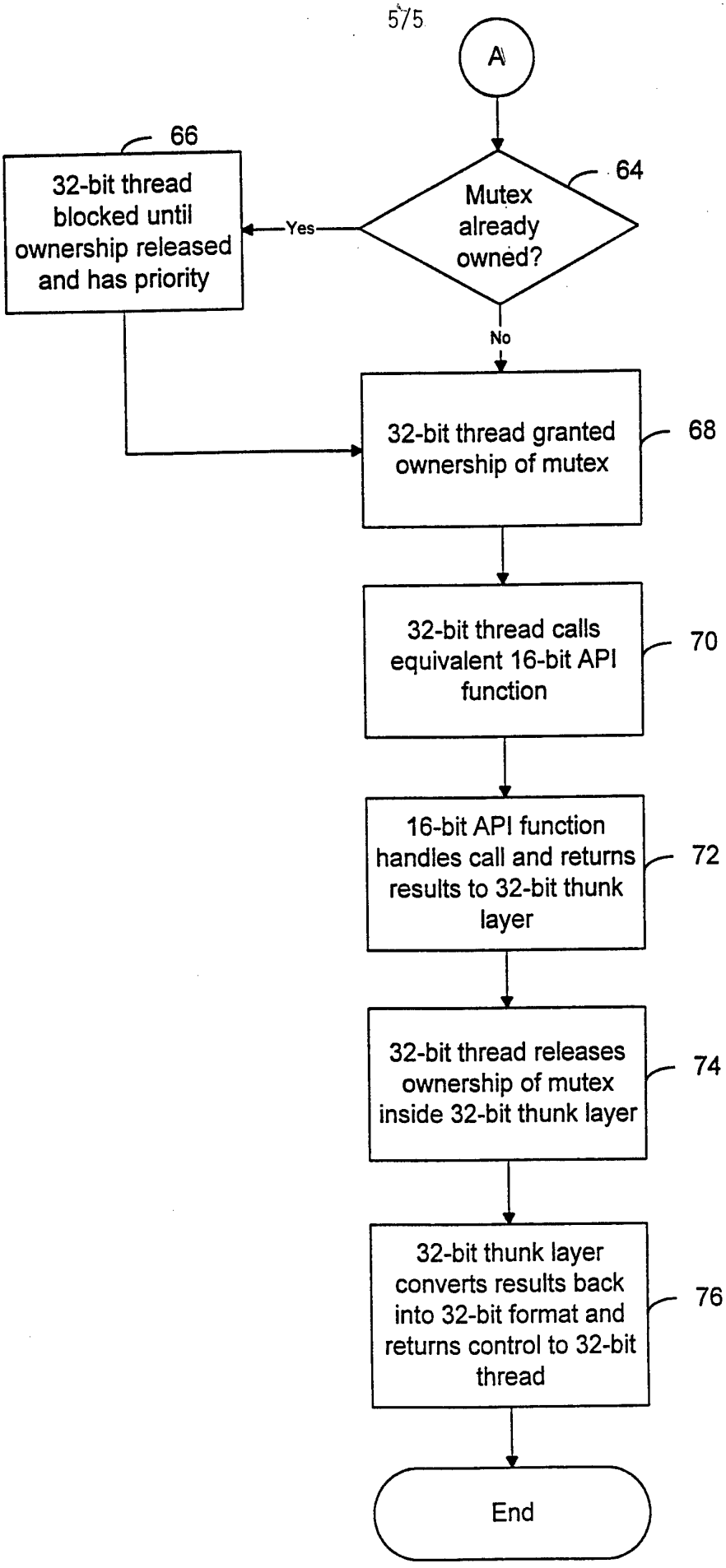

A
Mutex already owned?
64
Yes
No
32-bit thread blocked until ownership released and has priority
66
32-bit thread granted ownership of mutex
68
32-bit thread calls equivalent 16-bit API function
70
16-bit API function handles call and returns results to 32-bit thunk layer
72
32-bit thread releases ownership of mutex inside 32-bit thunk layer
74
32-bit thunk layer converts results back into 32-bit format and returns control to 32-bit thread
76
End

FIG. 4B

INTERNATIONAL SEARCH REPORT

Inte onal Application No
PCT/US 95/08287

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6 G06F9/46

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	REAL TIME SYSTEMS, vol. 3, no. 2, 1 May 1991 , NL, pages 149-163, XP 000306470 M. BOTTAZZI ET AL.: 'A Hierarchical Approach to Systems with Heterogeneous Real-Time Requirements'	1-4, 7-10, 13-15
Y	see page 150, line 15 - line 26 see page 156, line 28 - page 158, line 4 see page 152, line 31 - page 153, line 8 --- -/--	5,6,11, 12,16-21

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
11 October 1995	10. 11. 95

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax (+31-70) 340-3016	Wiltink, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inte .onal Application No
PCT/US 95/08287

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	IBM TECHNICAL DISCLOSURE BULLETIN, vol. 34, no. 4B, September 1991 , NEW YORK, US, pages 314-317, 'Sixteen to Thirty-two Bit Operating System Compatibility Method for Personal Computers.' see the whole document	5,6,11, 12,16-21
A	US,A,4 945 470 (TAKAHASHI) 31 July 1990 see abstract; figure 1 see column 1, line 29 - line 51 see column 1, line 66 - column 2, line 47	1,6,7, 12,13, 18,21
A	EP,A,0 381 655 (IBM) 8 August 1990 see abstract; claim 1 see page 4, line 1 - line 31	1,6,7, 12,13, 18,21

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Int. ional Application No
PCT/US 95/08287

Patent document cited in search report	Publication date	Patent family member(s)		Publication date
US-A-4945470	31-07-90	JP-A-	61006741	13-01-86
EP-A-381655	08-08-90	JP-C-	1856975	07-07-94
		JP-A-	2231640	13-09-90
		US-A-	5319782	07-06-94