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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
30 April 2009 (30.04.2009)

PCT

(10) International Publication Number
WO 2009/055494 A2

(51) International Patent Classification:
G06F 9/38 (2006.01)

(21) International Application Number:
PCT/US2008/080825

(22) International Filing Date: 22 October 2008 (22.10.2008)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
11/977,593 25 October 2007 (25.10.2007) US

(71) Applicant *(for all designated States except US)*: **MICROSOFT CORPORATION** [US/US]; One Microsoft Way, Redmond, WA 98052-6399 (US).

(72) Inventor: **CWALINA, Krysztof**; One Microsoft Way, Redmond, WA 98052-6399 (US).

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

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

Declarations under Rule 4.17:
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:
— *without international search report and to be republished upon receipt of that report*

(54) Title: TECHNIQUES FOR SWITCHING THREADS WITHIN ROUTINES

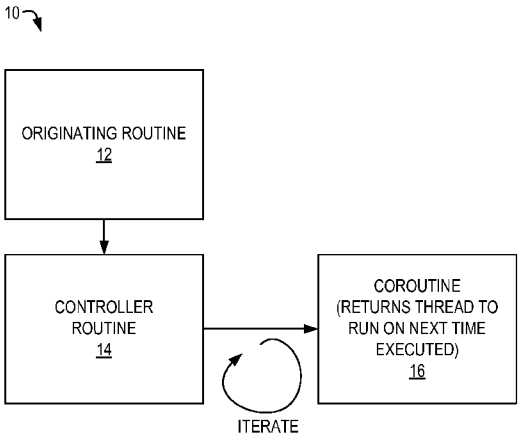

FIG. 1

(57) Abstract: Various technologies and techniques are disclosed for switching threads within routines. A controller routine receives a request from an originating routine to execute a coroutine, and executes the coroutine on an initial thread. The controller routine receives a response back from the coroutine when the coroutine exits based upon a return statement. Upon return, the coroutine indicates a subsequent thread that the coroutine should be executed on when the coroutine is executed a subsequent time. The controller routine executes the coroutine the subsequent time on the subsequent thread. The coroutine picks up execution at a line of code following the return statement. Multiple return statements can be included in the coroutine, and the threads can be switched multiple times using this same approach. Graphical user interface logic and worker thread logic can be co-mingled into a single routine.

WO 2009/055494 A2

TECHNIQUES FOR SWITCHING THREADS WITHIN ROUTINES

BACKGROUND

[0001] Developers write software applications using one or more software development programs. Developers write source code that is needed to make the software application perform the desired functionality. Software applications that have a user interface allow an end user to interact with graphical menus and options in the completed application to achieve a desired result. Source code generally has to be written by developers during development of the software application to handle such user input, and then to perform the proper work in response to the user input.

[0002] For example, in the case of a completed customer service application, the end user might be able to select a search operation to search for all customer records for a given customer name. The software application would then have to process the search, access a database to find the matching records, and return the results to the end user. If such a search is processed on the user interface thread of the software application, then an hourglass may or may not be displayed depending on the status of the processing. For example, an hourglass may not be displayed if the application is completely blocked. In such a blocking scenario, all that may be displayed on the screen is a black rectangle or other indicator which designates that the user interface thread is blocked. During this blocked period, the user is not able to do anything else with the program since the user interface thread is totally occupied in the performance of the search.

[0003] As technology has advanced, multi-threaded applications and multi-processor computers can now be utilized. In other words, multiple threads of execution can be started at the same time, sometimes on multiple processors when available. One thread, for example, may be used to handle user input, and another thread may be used for performing worker tasks. In order to create multi-threaded applications, developers are challenged with writing complex source code that creates and manages the multiple threads. This source code typically needs to include functionality for passing arguments between the different threads, which may be running asynchronously over many different locations. Developers often

write separate routines for the work that needs performed by separate threads. Due to the complexity of working with multiple threads, it is common for developers to use the multiple threads incorrectly or inefficiently, or to not even use multiple threads at all and just expect that users will be fine with infrequent user interface blocks.

SUMMARY

[0004] Various technologies and techniques are disclosed for switching threads within routines. A controller routine receives a request from an originating routine to execute a coroutine, and executes the coroutine on an initial thread. The controller routine receives a response back from the coroutine when the coroutine exits based upon a return statement. A subsequent thread is indicated that the coroutine should be executed on when the coroutine is executed a subsequent time. The controller routine executes the coroutine the subsequent time on the subsequent thread that was previously indicated. Multiple return statements can be included in the coroutine, and these steps can be repeated multiple times to switch threads.

[0005] In one implementation, execution of graphical user interface logic and worker thread logic can be co-mingled into a single coroutine. Code execution starts for initial logic contained in a coroutine, with the initial logic being executed on a first thread to receive user input. The user input is then received. The coroutine returns partially through to switch to a second thread, with the second thread being a different thread than the first thread. Code execution resumes in the coroutine on the second thread to perform work in response to the user input.

[0006] This Summary was provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used as an aid in determining the scope of the claimed subject matter.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] Figure 1 is a diagrammatic view of a thread switching system of one implementation.

[0008] Figure 2 is a diagrammatic view of a computer system of one implementation.

[0009] Figure 3 is a diagrammatic view of a thread switching controller application of one implementation.

[0010] Figure 4 is a process flow diagram for one implementation illustrating the stages involved in using a controller routine to manage execution of a coroutine on different threads.

[0011] Figure 5 is a process flow diagram for one implementation illustrating the stages involved in switching threads partially through a routine.

[0012] Figure 6 is a process flow diagram for one implementation illustrating the stages involved in enabling graphical user interface logic and worker thread logic to be co-mingled into a single routine.

[0013] Figure 7 illustrates exemplary source code for an OnClick event associated with a form of a user interface to show an example of how a controller routine is started.

[0014] Figure 8 illustrates exemplary source code for a controller routine that manages the execution of a coroutine on multiple threads.

[0015] Figure 9 illustrates exemplary source code for a coroutine that co-mingles graphical user interface logic and worker thread logic into a single coroutine.

DETAILED DESCRIPTION

[0016] The technologies and techniques herein may be described in the general context as techniques that enable switching between threads within a single routine, but the technologies and techniques also serve other purposes in addition to these. In one implementation, one or more of the techniques described herein can be implemented as features within a software development program such as MICROSOFT® VISUAL STUDIO®, from any type of program or service that is used to write source code for a software application, or from any other type of program or service that is used to create or manage multi-threaded software applications.

[0017] In one implementation, coroutines can be used with some or all of the technologies and techniques described herein to enable code to be written that

appears to be sequential and passes arguments with a regular language syntax, but that also allows for switching between multiple threads that execute the logic contained in the single routine. In other words, the technologies and techniques described herein provide mechanisms for co-mingling the usage of logic that is to be executed on different threads within a same routine. The term "coroutine" as used herein is meant to include a function, procedure, or other routine that contains a set of code statements that are co-located within the coroutine and allows multiple entry points that can be suspended and resumed at later times, with the lifetime of a particular activation record of the coroutine being independent of the time when control enters or leaves the coroutine

[0018] Figure 1 is a diagrammatic view of a thread switching system 10 of one implementation. In one implementation, thread switching system 10 includes an originating routine 12, a controller routine 14, and a coroutine 16. The term "routine" as used herein is meant to include a program component, including a function, procedure, or any other manner of grouping source code into a unit. An originating routine 12 can be any routine that wants to execute functionality that is contained in a given coroutine, such as coroutine 16. In one implementation, in order to call the coroutine, the originating routine goes through a controller routine 14. The controller routine 14 iteratively calls the coroutine 16 multiple times, each time starting the coroutine 16 on a thread that was indicated by the coroutine when the coroutine returned. The term "return statement" as used herein is meant to include a statement or other mechanism for causing a coroutine to return before an end of the coroutine is reached. In other words, when the coroutine 16 wishes to switch threads before executing any more lines of code, the return statement is used to return from the routine temporarily so a different thread can be used to process the lines of code that follow. There are various ways that the coroutine can indicate the subsequent thread upon returning. In one implementation, the return statement can include a return parameter that includes a subsequent thread identifier for the different thread that should be used next. Various other ways for indicating the subsequent thread can also be used, such as by the coroutine calling a method to set the subsequent thread before returning, by setting a property or value in an object

with the subsequent thread identifier, by recording the subsequent thread identifier to a database, etc.

[0019] The coroutine 16 is then resumed (on the different thread) at the line of code following the return statement (one or more lines later), or at another suitable location. The stages can be repeated until the end of the coroutine 16 is reached, or another event happens that causes the controller routine 14 to stop iterating through execution of the coroutine 16 and switching threads. In one implementation, each time the coroutine 16 is resumed by the controller routine 14, a most recent subsequent thread that was indicated in a most recent return from the coroutine can be used to determine the thread to use.

[0020] It should be noted that in another implementation, the functionality of the originating routine 12 and the controller routine 14 can be combined into the same routine. In some implementations, the term "routine" as used herein can also include coroutines. For example, while originating routine 12 and controller routine 14 are both described as routines, in some implementations, either or both could be implemented as one or more coroutines. These techniques introduced in the discussion of Figure 1 will be described in much greater detail in Figures 3-6, and then with source code examples in Figures 7-9.

[0021] Turning now to Figure 2, an exemplary computer system to use for implementing one or more parts of the system is shown that includes a computing device, such as computing device 100. In its most basic configuration, computing device 100 typically includes at least one processing unit 102 and memory 104. Depending on the exact configuration and type of computing device, memory 104 may be volatile (such as RAM), non-volatile (such as ROM, flash memory, etc.) or some combination of the two. This most basic configuration is illustrated in Figure 2 by dashed line 106.

[0022] Additionally, device 100 may also have additional features/functionality. For example, device 100 may also include additional storage (removable and/or non-removable) including, but not limited to, magnetic or optical disks or tape. Such additional storage is illustrated in Figure 2 by removable storage 108 and non-removable storage 110. Computer storage media includes volatile and nonvolatile,

removable and non-removable media implemented in any method or technology for storage of information such as computer readable instructions, data structures, program modules or other data. Memory 104, removable storage 108 and non-removable storage 110 are all examples of computer storage media. Computer storage media includes, but is not limited to, RAM, ROM, EEPROM, flash memory or other memory technology, CD-ROM, digital versatile disks (DVD) or other optical storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium which can be used to store the desired information and which can accessed by device 100. Any such computer storage media may be part of device 100.

[0023] Computing device 100 includes one or more communication connections 114 that allow computing device 100 to communicate with other computers/ applications 115. Device 100 may also have input device(s) 112 such as keyboard, mouse, pen, voice input device, touch input device, etc. Output device(s) 111 such as a display, speakers, printer, etc. may also be included. These devices are well known in the art and need not be discussed at length here. In one implementation, computing device 100 includes thread switching controller application 200. Thread switching controller application 200 will be described in further detail in Figure 3.

[0024] Turning now to Figure 3 with continued reference to Figure 2, a thread switching controller application 200 operating on computing device 100 is illustrated. Thread switching controller application 200 is one of the application programs that reside on computing device 100. However, it will be understood that thread switching controller application 200 can alternatively or additionally be embodied as computer-executable instructions on one or more computers and/or in different variations than shown on Figure 2. Alternatively or additionally, one or more parts of thread switching controller application 200 can be part of system memory 104, on other computers and/or applications 115, or other such variations as would occur to one in the computer software art.

[0025] Thread switching controller application 200 includes program logic 204, which is responsible for carrying out some or all of the techniques described herein. Program logic 204 includes logic for receiving a request from the originating

routine to execute a coroutine 206 (as described below with respect to Figure 4); logic for executing the coroutine on an initial thread 208 (as described below with respect to Figure 4); logic for receiving a response back from the coroutine when the coroutine returns 210 (as described below with respect to Figure 4); logic for executing the coroutine on a subsequent thread indicated when the coroutine returned 212 (as described below with respect to Figure 4); logic for enabling graphical user interface logic and worker thread logic to be co-mingled into a single routine 214 (as described below with respect to Figure 6); and other logic 220 for operating the thread switching controller application 200.

[0026] Turning now to Figures 4-6 with continued reference to Figures 1-3, the stages for implementing one or more implementations of thread switching system 10 (of Figure 1) and/or thread switching controller application 200 (of Figure 3) are described in further detail. In some implementations, the processes of Figure 4-6 are at least partially implemented in the operating logic of computing device 100.

[0027] Figure 4 is a process flow diagram 240 that illustrates one implementation of the stages involved in using controller routine to manage execution of a coroutine on different threads. In general, Figure 4 describes an exemplary process involved in having the controller routine call the coroutine multiple times, each time calling the coroutine on a thread that was specified when the coroutine returned based upon a return statement.

[0028] When a request is received from an originating routine to execute a coroutine (stage 242), the controller routine executes the coroutine on an initial thread (stage 244). In one implementation, before a line of code is encountered in the coroutine to initiate the switching to a desired thread, the initial thread will simply be a default thread that is used by the software application when a thread is not otherwise specified. The coroutine then executes some logic, and the controller routine receives a response back from the coroutine based upon a return statement (stage 246). Upon exit, the coroutine had indicated a subsequent thread identifier to specify what thread the coroutine should be called on the next time the controller routine calls the coroutine. In one implementation, the response received back from the coroutine from the return statement contains a parameter with the

subsequent thread identifier. In other implementations, the controller routine accesses the value in an object, database, or other variable that was assigned the subsequent thread identifier when the coroutine returned.

[0029] The controller routine then executes the coroutine a subsequent time on the thread that was indicated when the coroutine returned (i.e. indicated in the subsequent thread identifier) (stage 248). If the coroutine returns again before the end of the coroutine is reached (i.e. because of another return statement) (decision point 250), then the controller routine will receive another response back from the coroutine and will have an indication of the next subsequent thread to start the coroutine on the next time it is called (stage 246). The controller routine will then execute the coroutine again on the next subsequent thread (stage 248). Once the end of the coroutine is reached (decision point 250), the controller routine stops calling the coroutine (stage 252).

[0030] Figure 5 is a process flow diagram 270 that illustrates one implementation of the stages involved switching threads partially through a routine. The routine is started on one thread (stage 272). In one implementation, the routine is a coroutine. In another implementation, the routine is any type of routine that can be called multiple times, and that is operable to have execution started at a point in the code where the prior execution stopped. After executing one or more lines of code in the routine, the routine encounters a return statement, thereby causing the routine to return at a point only partially through all the code in the routine (stage 274). Upon returning, the routine causes a subsequent thread identifier to be indicated (stage 274). There are various ways that the routine can cause the subsequent thread identifier to be indicated, such as in a return parameter of the return statement itself, by setting a property or field of an object or type to set the value, by calling a method that sets the value, by writing the subsequent thread identifier to a database, or through any other means of communicating the subsequent thread identifier to the controller routine. Execution of the routine is then resumed on the thread that was specified in the subsequent thread identifier (stage 276). Code execution is picked up where it left off the last time before return, this time on the subsequent thread (stage 278).

[0031] Figure 6 is a process flow diagram 290 that illustrates one implementation of the stages involved in enabling graphical user interface thread logic and worker thread logic to be co-mingled into a single routine. A coroutine is started on a first thread to receive user input (stage 292). The user input is then received (stage 294). The coroutine returns partially through to switch to a second thread (stage 296), with the second thread being different than the first thread. In one implementation, the coroutine returns because it encountered a return statement. Code execution is resumed in the coroutine on the second thread to do work in response to the user input (stage 298). In one implementation, the first thread is a graphical user interface thread, and the second thread is a worker thread. Various thread combinations are possible, instead of or in addition to graphical user interface threads and/or worker threads. Figure 9 shows exemplary source code of co-mingling graphical user interface thread logic and worker thread logic into the same routine.

[0032] Turning now to Figures 7-9, exemplary source code will be described to illustrate the stages of Figures 4-6 in further detail. Beginning with Figure 7, exemplary source code 310 is shown for an OnClick event 312 associated with a form of a user interface. The OnClick event 312 (originating routine 12 on Figure 1) fires when an end user clicks the BetterButton button on the form. The OnClick event 312 then calls the controller routine (14 on Figure 1), which is the execute method 314 in the example shown. In one implementation, the name of the coroutine that needs to be executed by the controller routine is passed as a parameter to the controller routine from the originating routine. The controller routine is then responsible for calling the coroutine multiple times on the proper threads, as is indicated in Figure 8. In the exemplary controller routine 322 shown in Figure 8, various lines of source code are included for handling work on multiple threads. Calls to the coroutine are also included within a loop 324, which is responsible for calling the coroutine multiple times. Each time the coroutine is called, the controller routine starts the coroutine on the thread that was specified previously when the coroutine returned.

[0033] As shown in Figure 9, the coroutine can include code that is designed to be executed on different types of threads mixed into a single coroutine. In the exemplary source code 330 of Figure 9, graphical user interface logic (336 and 344) and worker thread logic (340) are co-mingled together into a single routine, which is the betterButton1_Clicked routine 332 in this example. In order to switch between threads, return statements (334, 338, and 342) are used. In this example, GUI threads and worker threads are used for executing separate parts of work. Furthermore, in this example, the return statement is a yield statement (for the C# programming language). Each return statement (334, 338, and 342) signals that the coroutine should be returned temporarily and then resumed in the thread specified as a parameter to the return statement (e.g. "ThreadRequest.UI" for the GUI thread, or "ThreadRequest.Worker" for the worker thread). When the controller routine shown in Figure 8 receives the return response from the coroutine, the execution of the coroutine is then started again by the controller routine, but on the specified thread.

[0034] Although the subject matter has been described in language specific to structural features and/or methodological acts, it is to be understood that the subject matter defined in the appended claims is not necessarily limited to the specific features or acts described above. Rather, the specific features and acts described above are disclosed as example forms of implementing the claims. All equivalents, changes, and modifications that come within the spirit of the implementations as described herein and/or by the following claims are desired to be protected.

[0035] For example, a person of ordinary skill in the computer software art will recognize that the examples discussed herein could be organized differently on one or more computers to include fewer or additional options or features than as portrayed in the examples.

CLAIMS

What is claimed is:

1. A computer-readable medium having computer-executable instructions for causing a computer to perform steps comprising:

receiving a request from an originating routine to execute a coroutine (206);

executing the coroutine on an initial thread (208);

receiving a response back from the coroutine when the coroutine returns based upon a return statement (210); and

executing the coroutine a subsequent time on a subsequent thread that was indicated by the coroutine upon return (212).

2. The computer-readable medium of claim 1, further having computer-executable instructions for causing a computer to perform steps comprising:

executing the coroutine a plurality of times until an end of the coroutine is reached, each time executing the coroutine on a most recent subsequent thread indicated by the coroutine upon a most recent return (250).

3. The computer-readable medium of claim 1, wherein the subsequent thread is contained as a parameter to the return statement (246).

4. The computer-readable medium of claim 1, wherein the subsequent thread is used for processing worker thread logic (296).

5. A method for switching threads partially through a routine comprising the steps of:

starting an execution of a routine on a first thread (272);

exiting the routine at a point partially through the routine due to a first return statement, with the first return statement having a subsequent thread identifier (274);

resuming code execution of the routine on a subsequent thread specified in the subsequent thread identifier (276); and

picking up code execution in the routine at a line of code following the first return statement (278).

6. The method of claim 5, wherein at least one of the first thread and the subsequent thread is a thread used for processing graphical user interface logic (292).

7. The method of claim 5, wherein at least one of the first thread and the subsequent thread is a worker thread used for processing worker logic (298).

8. The method of claim 5, wherein the routine is a coroutine (272).

9. The method of claim 8, wherein the first thread is used to handle a different type of operation than the second thread (298).

10. The method of claim 5, wherein the subsequent thread identifier is contained as a parameter in the return statement (274).

11. The method of claim 5, further comprising:

repeating the exiting, resuming, and picking up steps a plurality of times until an end of the routine is reached (250).

12. The method of claim 5, wherein a first line of code contained in the routine is the first return statement (334).

13. The method of claim 12, wherein the first return statement is located at the first line of code so that a first desired thread for execution can be specified in the subsequent thread identifier before further code in the routine is processed (334).

14. A method for co-mingling graphical user interface logic and worker thread logic in a single routine comprising the steps of:

starting code execution of initial logic contained in a coroutine, the initial logic being executed on a first thread to receive user input (292);

receiving the user input (294);

exiting the coroutine partially through to switch to a second thread, the second thread being a different thread than the first thread (296); and

resuming code execution in the coroutine on the second thread to perform work in response to the user input (298).

15. The method of claim 14, wherein the exiting step is performed when a return statement is reached (296).

16. The method of claim 15, wherein the return statement contains a thread identifier as a parameter to specify a next thread that execution should be resumed on (274).

17. The method of claim 16, wherein a plurality of return statements are contained in the routine to indicate when thread switching should occur (16).

18. The method of claim 14, wherein the coroutine is called a plurality of times by a controller routine that is responsible for calling the coroutine on a proper thread (14).

19. The method of claim 18, wherein the controller routine is called by an originating routine (12).

20. The method of claim 19, wherein the originating routine contains a request to call the coroutine (12).

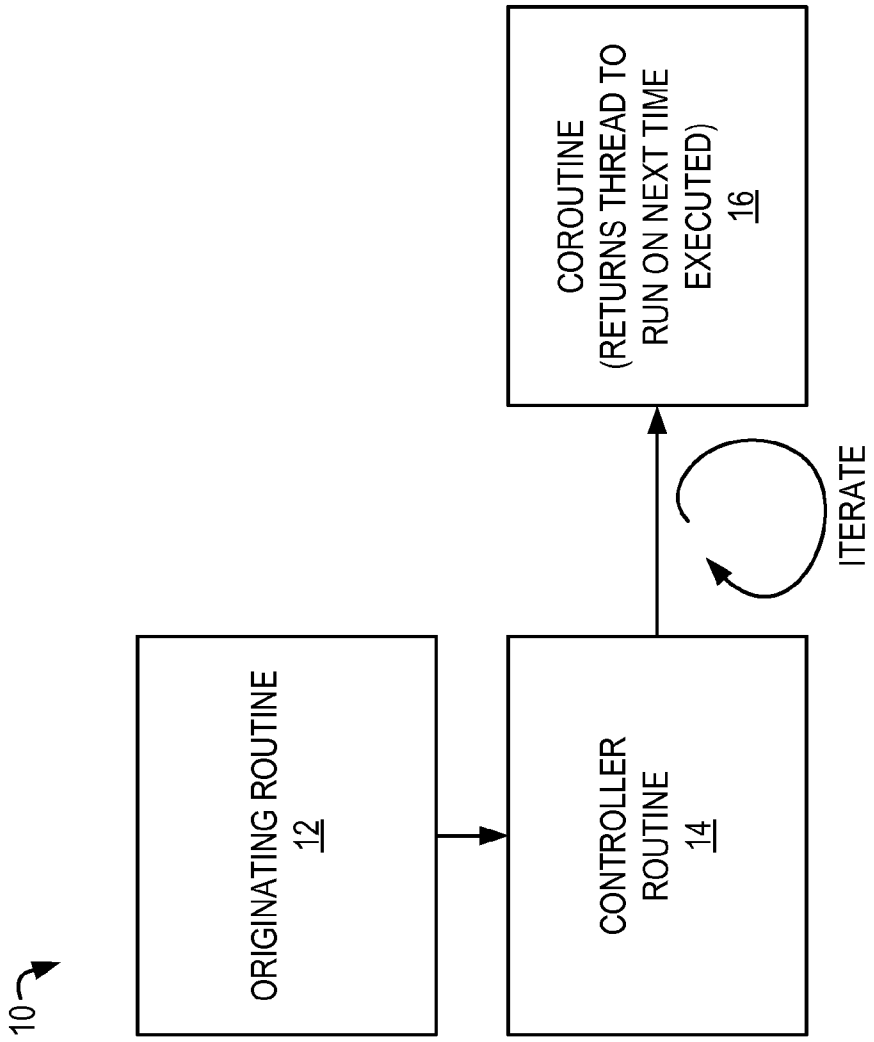
10
ORIGINATING ROUTINE
12
CONTROLLER ROUTINE
14
ITERATE
COROUTINE (RETURNS THREAD TO RUN ON NEXT TIME EXECUTED)
16

FIG. 1

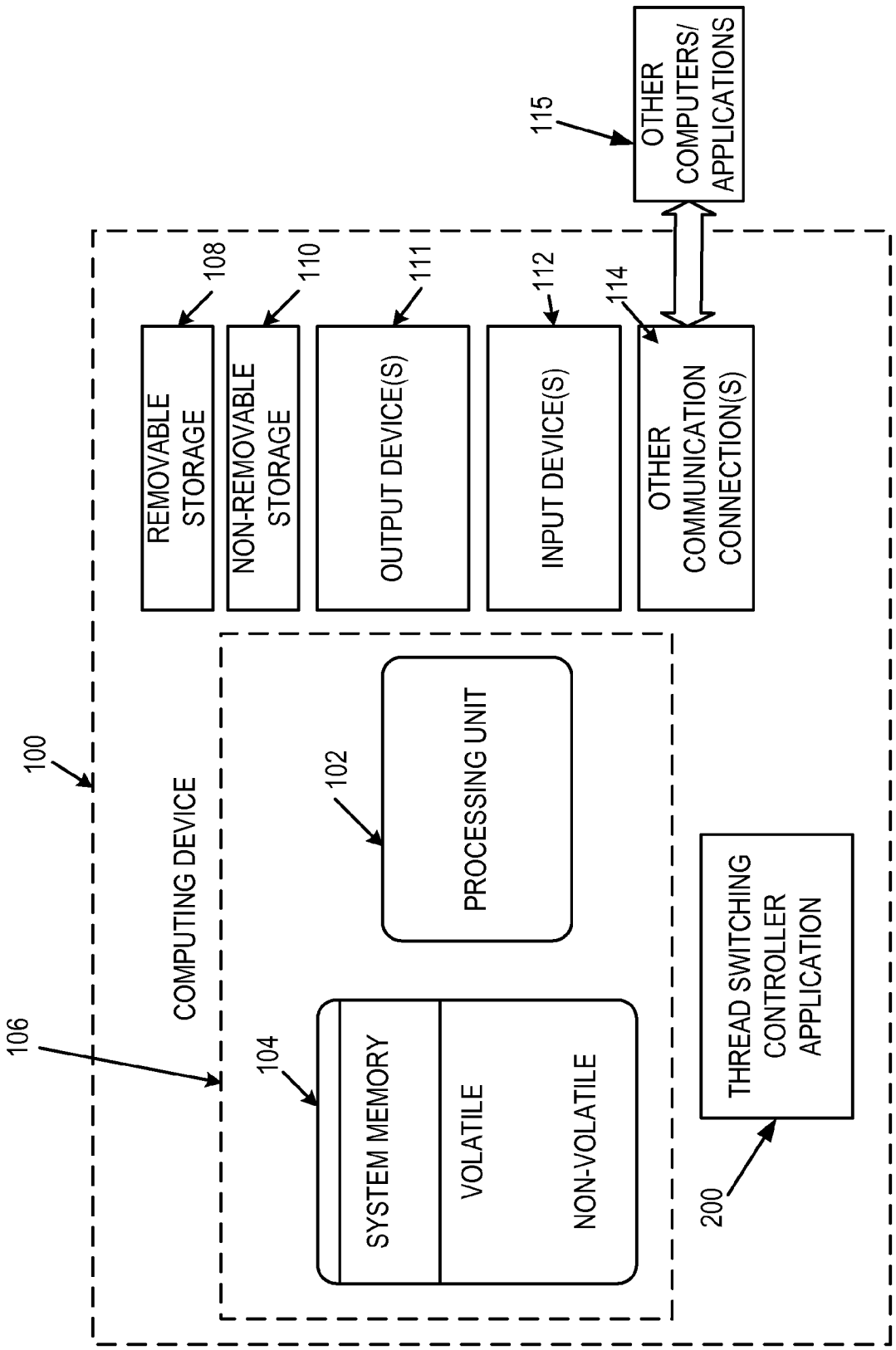
100
COMPUTING DEVICE
106
102
PROCESSING UNIT
104
SYSTEM MEMORY
VOLATILE
NON-VOLATILE
200
THREAD SWITCHING CONTROLLER APPLICATION
108
REMOVABLE STORAGE
110
NON-REMOVABLE STORAGE
111
OUTPUT DEVICE(S)
112
INPUT DEVICE(S)
114
OTHER COMMUNICATION CONNECTION(S)
115
OTHER COMPUTERS/ APPLICATIONS

FIG. 2

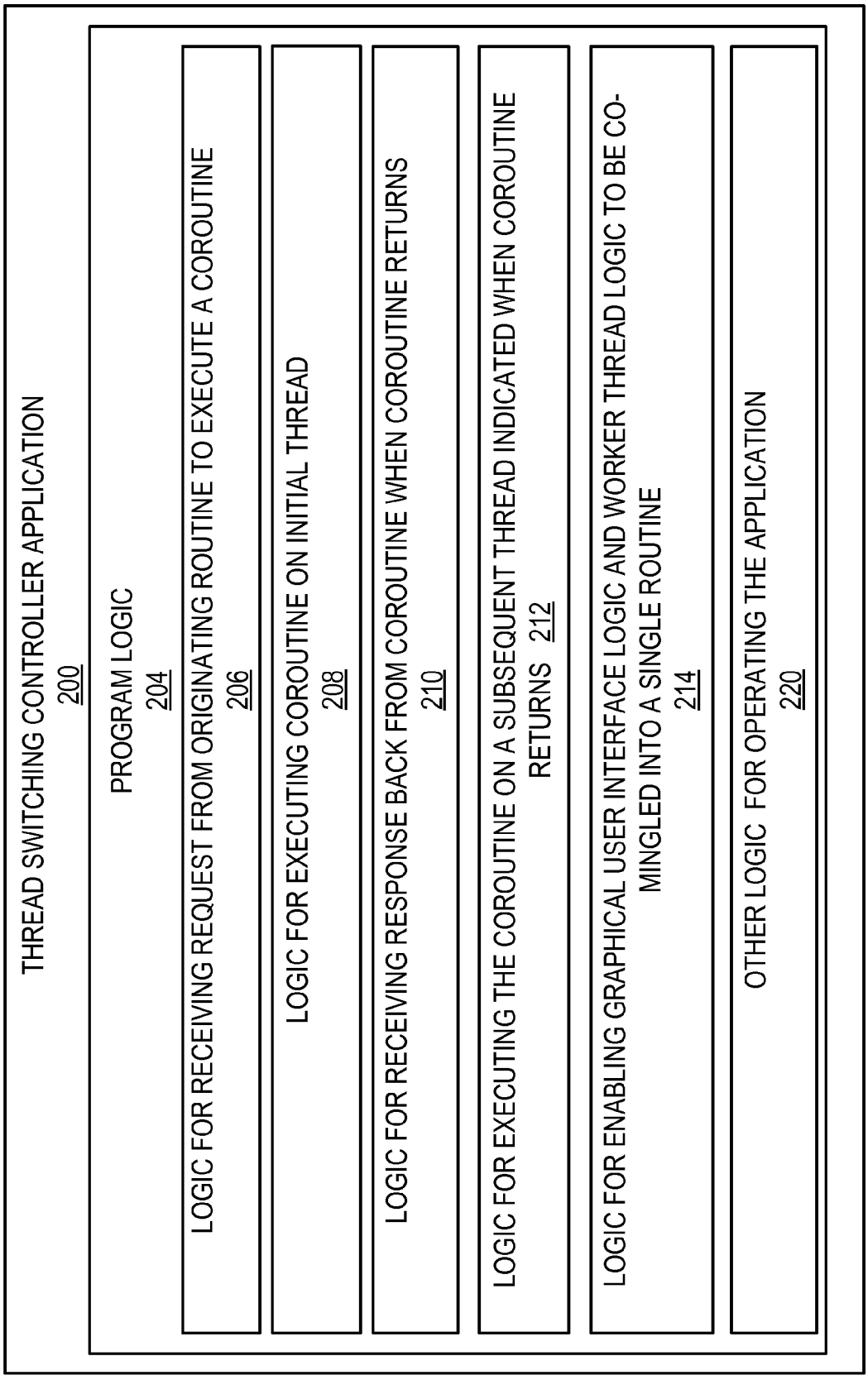
THREAD SWITCHING CONTROLLER APPLICATION
200
PROGRAM LOGIC
204
LOGIC FOR RECEIVING REQUEST FROM ORIGINATING ROUTINE TO EXECUTE A COROUTINE
206
LOGIC FOR EXECUTING COROUTINE ON INITIAL THREAD
208
LOGIC FOR RECEIVING RESPONSE BACK FROM COROUTINE WHEN COROUTINE RETURNS
210
LOGIC FOR EXECUTING THE COROUTINE ON A SUBSEQUENT THREAD INDICATED WHEN COROUTINE RETURNS 212
LOGIC FOR ENABLING GRAPHICAL USER INTERFACE LOGIC AND WORKER THREAD LOGIC TO BE CO-MINGLED INTO A SINGLE ROUTINE
214
OTHER LOGIC FOR OPERATING THE APPLICATION
220

FIG. 3

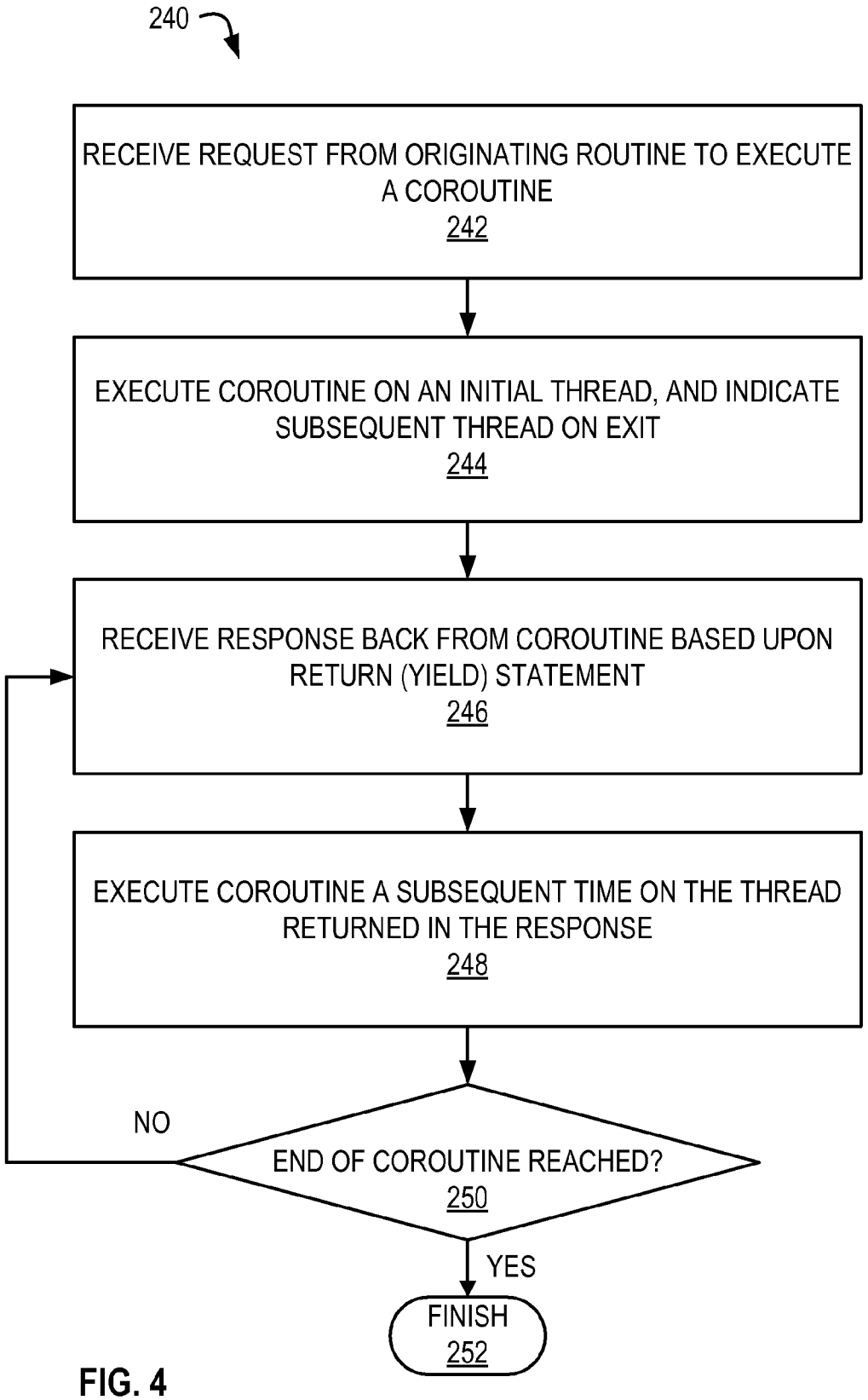
240
RECEIVE REQUEST FROM ORIGINATING ROUTINE TO EXECUTE A COROUTINE
242
EXECUTE COROUTINE ON AN INITIAL THREAD, AND INDICATE SUBSEQUENT THREAD ON EXIT
244
RECEIVE RESPONSE BACK FROM COROUTINE BASED UPON RETURN (YIELD) STATEMENT
246
EXECUTE COROUTINE A SUBSEQUENT TIME ON THE THREAD RETURNED IN THE RESPONSE
248
NO
END OF COROUTINE REACHED?
250
YES
FINISH
252

FIG. 4

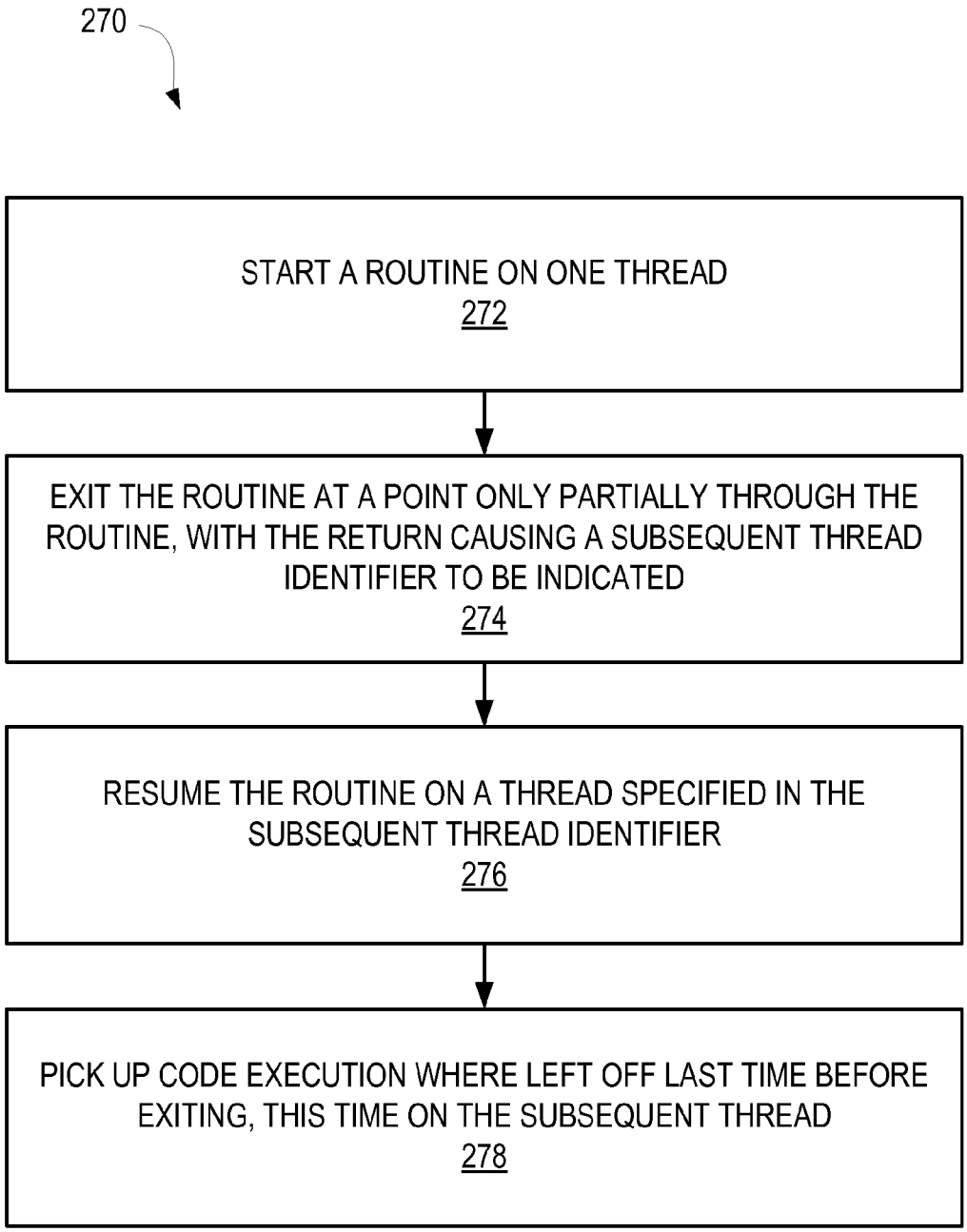
270
START A ROUTINE ON ONE THREAD
272
EXIT THE ROUTINE AT A POINT ONLY PARTIALLY THROUGH THE ROUTINE, WITH THE RETURN CAUSING A SUBSEQUENT THREAD IDENTIFIER TO BE INDICATED
274
RESUME THE ROUTINE ON A THREAD SPECIFIED IN THE SUBSEQUENT THREAD IDENTIFIER
276
PICK UP CODE EXECUTION WHERE LEFT OFF LAST TIME BEFORE EXITING, THIS TIME ON THE SUBSEQUENT THREAD
278

FIG. 5

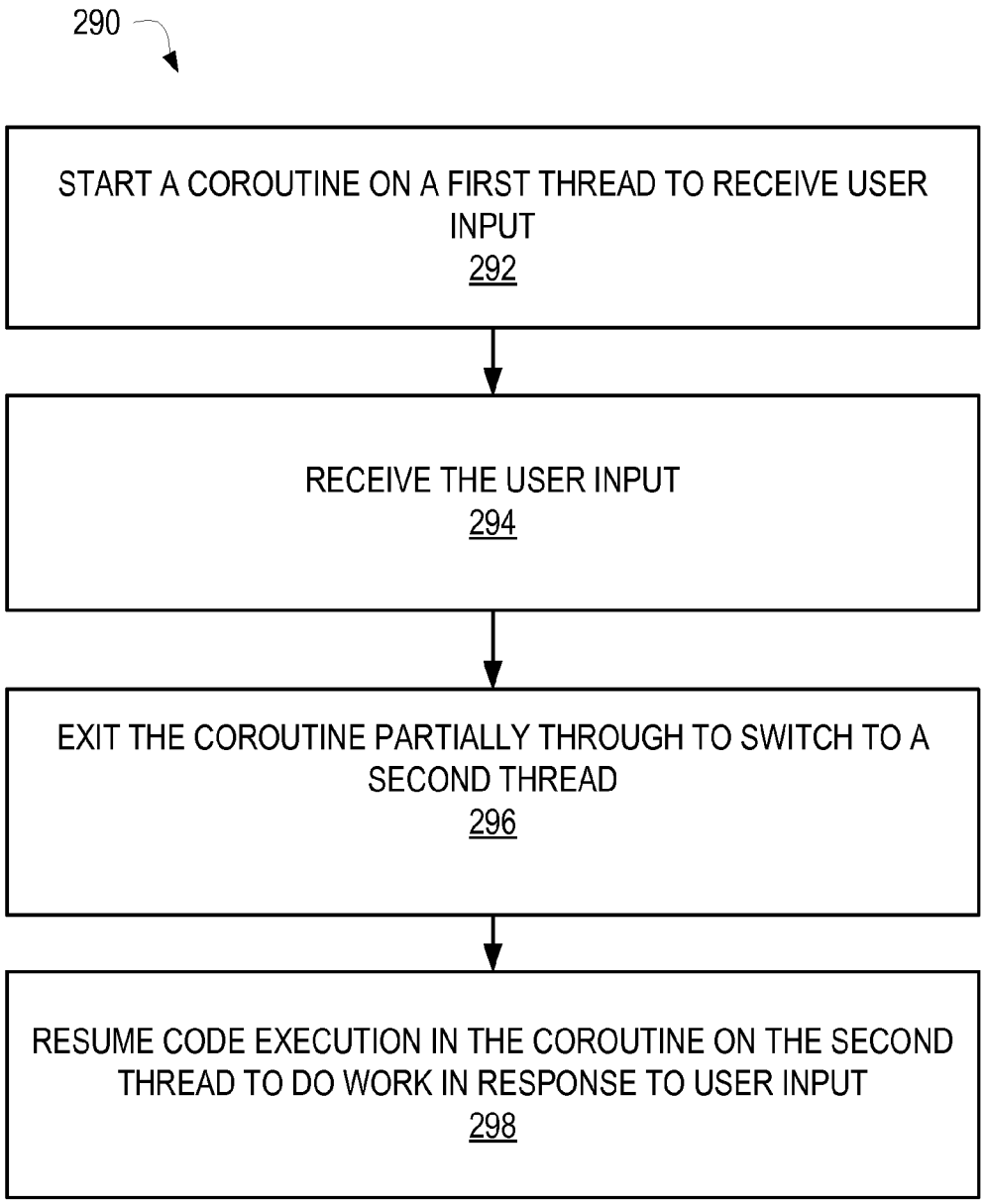
290
START A COROUTINE ON A FIRST THREAD TO RECEIVE USER INPUT
292
RECEIVE THE USER INPUT
294
EXIT THE COROUTINE PARTIALLY THROUGH TO SWITCH TO A SECOND THREAD
296
RESUME CODE EXECUTION IN THE COROUTINE ON THE SECOND THREAD TO DO WORK IN RESPONSE TO USER INPUT
298

FIG. 6

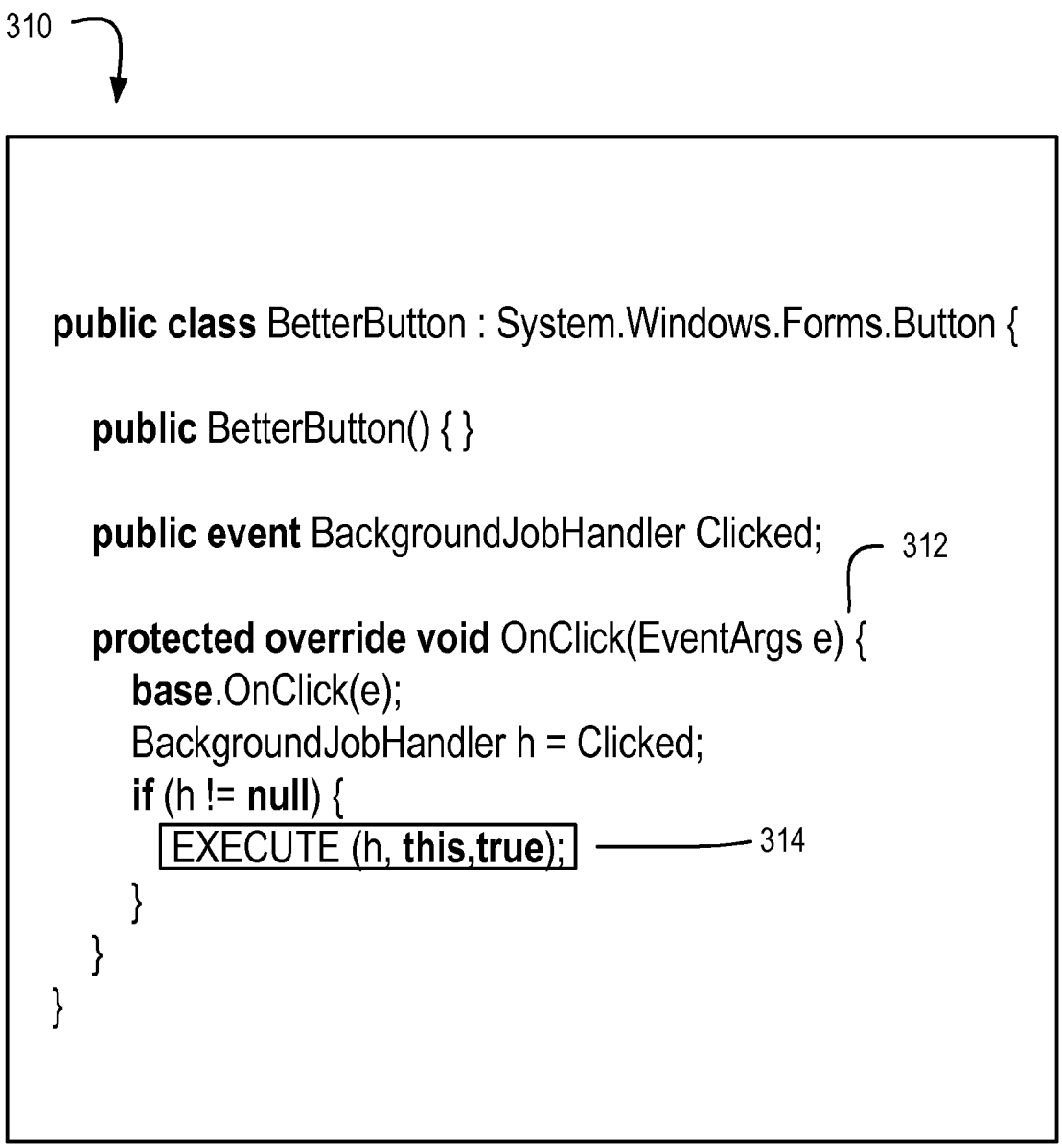
310
public class BetterButton : System.Windows.Forms.Button {
public BetterButton() { }
public event BackgroundJobHandler Clicked;
312
protected override void OnClick(EventArgs e) {
base.OnClick(e);
BackgroundJobHandler h = Clicked;
if (h != null) {
EXECUTE (h, this,true);
314
}
}
}

FIG. 7

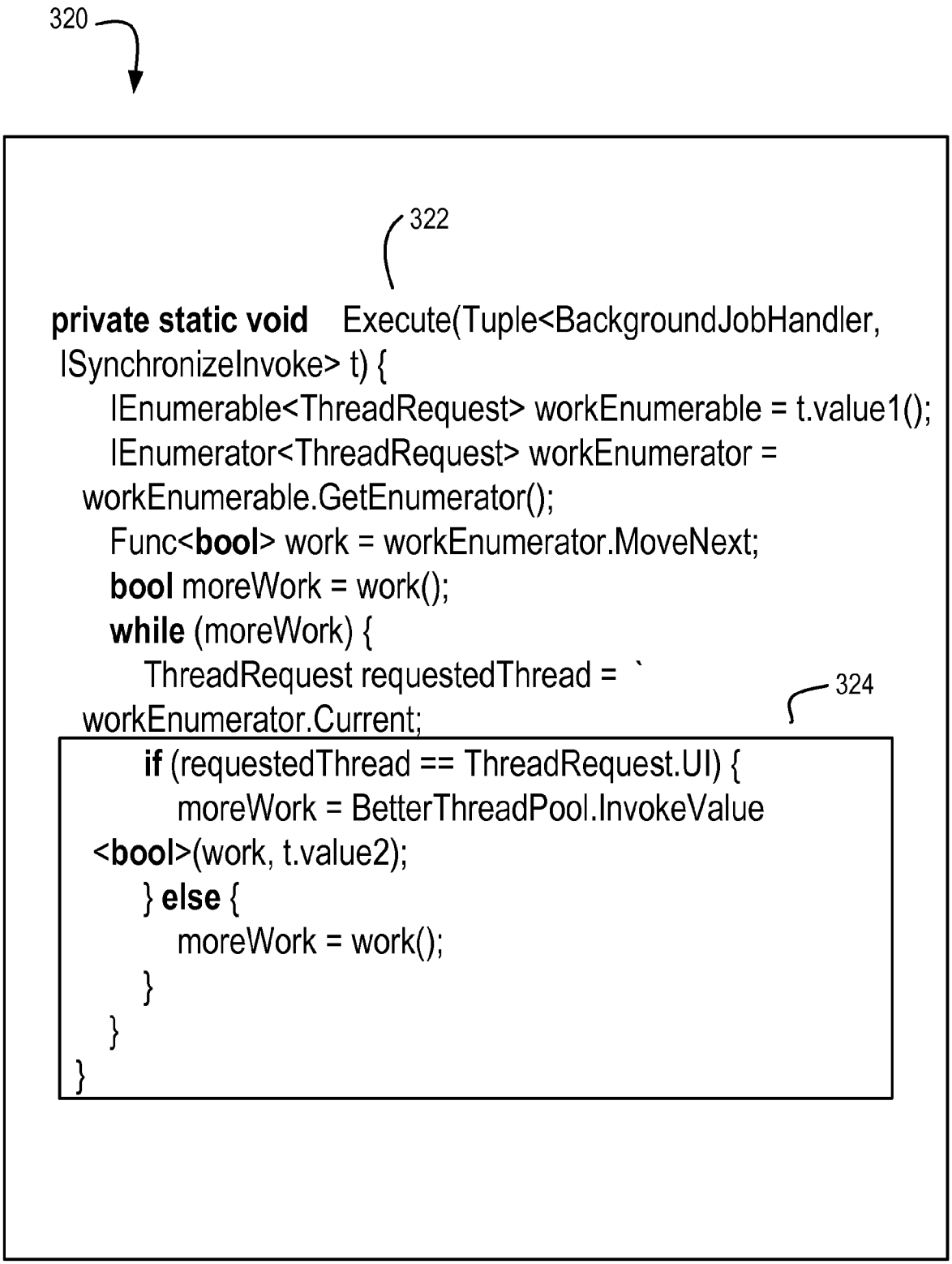
320
322
private static void Execute(Tuple<BackgroundJobHandler,
ISynchronizeInvoke> t) {
IEnumerable<ThreadRequest> workEnumerable = t.value1();
IEnumerator<ThreadRequest> workEnumerator =
workEnumerable.GetEnumerator();
Func<bool> work = workEnumerator.MoveNext;
bool moreWork = work();
while (moreWork) {
ThreadRequest requestedThread = `
workEnumerator.Current;
324
if (requestedThread == ThreadRequest.UI) {
moreWork = BetterThreadPool.InvokeValue
<bool>(work, t.value2);
} else {
moreWork = work();
}
}
}

FIG. 8

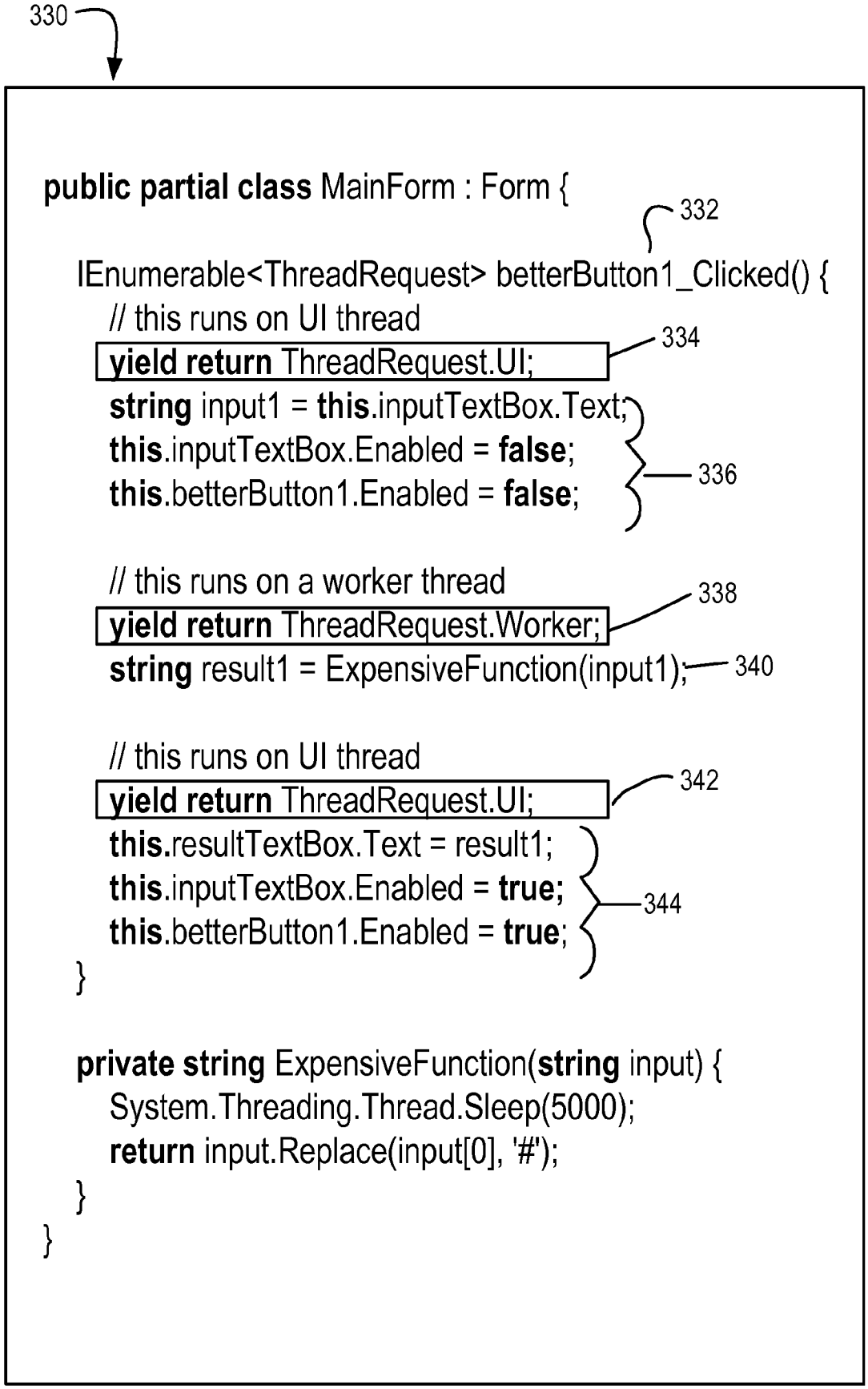
330
public partial class MainForm : Form {
332
IEnumerable<ThreadRequest> betterButton1_Clicked() {
// this runs on UI thread
yield return ThreadRequest.UI;
334
string input1 = this.inputTextBox.Text;
this.inputTextBox.Enabled = false;
this.betterButton1.Enabled = false;
336
// this runs on a worker thread
yield return ThreadRequest.Worker;
338
string result1 = ExpensiveFunction(input1);
340
// this runs on UI thread
yield return ThreadRequest.UI;
342
this.resultTextBox.Text = result1;
this.inputTextBox.Enabled = true;
this.betterButton1.Enabled = true;
344
}
private string ExpensiveFunction(string input) {
System.Threading.Thread.Sleep(5000);
return input.Replace(input[0], '#');
}
}

FIG. 9