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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 April 2005 (07.04.2005)

PCT

(10) International Publication Number
WO 2005/031569 A1

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number:
PCT/US2004/031371

(22) International Filing Date:
27 September 2004 (27.09.2004)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
10/670,802 26 September 2003 (26.09.2003) US

(71) Applicant *(for all designated States except US)*: **FINITE STATE MACHINE LABS, INC.** [US/US]; 115-D Abeyta Avenue, P.O. Box 1822, Socorro, NM 87801 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **YODAIKEN, Victor** [US/US]; 914 Paisano Drive, Socorro, NM 87801 (US).

DOUGAN, Cort [US/US]; 115-D Abeyta Avenue, P.O. Box 1822, Socorro, NM 87801 (US).

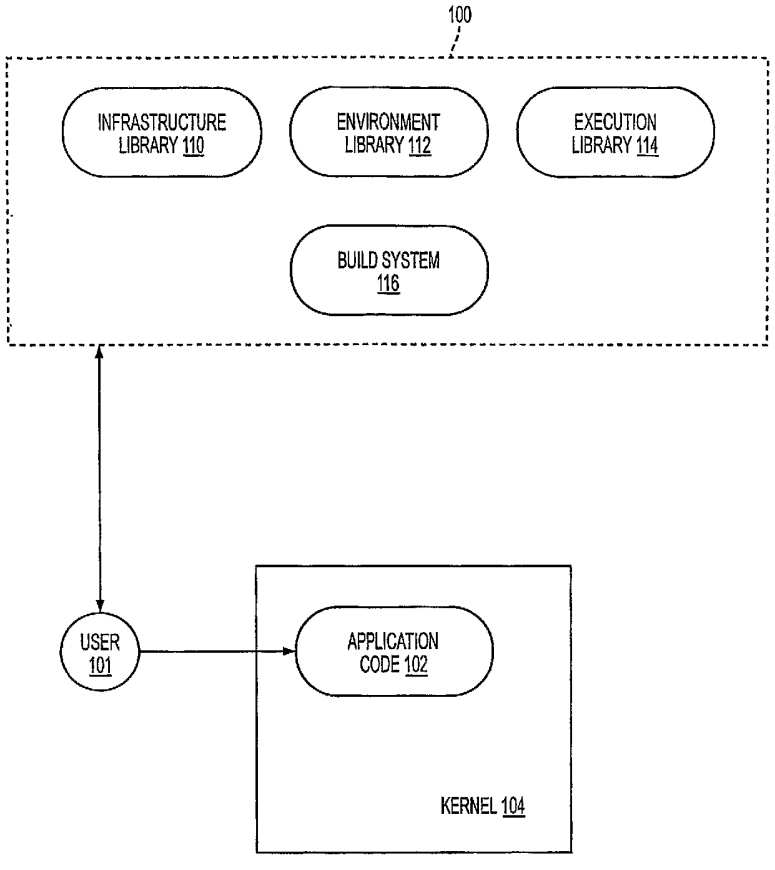
(74) Agents: **ZOLTICK, Martin, M.** et al.; 1425 K Street, N.W., Suite 800, Washington, DC 20005 (US).

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

[Continued on next page]

(54) Title: SYSTEMS AND METHODS FOR DYNAMICALLY LINKING APPLICATION SOFTWARE INTO A RUNNING OPERATING SYSTEM KERNEL



(57) Abstract: The present invention provides systems and methods for dynamically linking modules into a running operating system kernel. Systems and methods described herein have the following advantages: they permit an application programmer (101) to write, compile, execute, and terminate application code (102) that is to be loaded into a kernel (104) as if the application code was an ordinary application program, they allow a standard programming environment (100) to be used to encapsulate application software in a familiar environment, and they permit automatic cleanup of errors and freeing of program resources when the application terminates.

WO 2005/031569 A1

European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

Published:

— *with international search report*

SYSTEMS AND METHODS FOR DYNAMICALLY LINKING APPLICATION SOFTWARE INTO A RUNNING OPERATING SYSTEM KERNEL

COPYRIGHT NOTICE

COMPUTER PROGRAM LISTING APPENDIX

[002] A computer program listing appendix on a compact disc incorporating features of the present invention has been submitted with the earlier national application, U.S. Application S.N. 10/670,802, in which this patent application claims priority, and is incorporated herein by reference in its entirety. The files contained on the disc are: (1) user.c (size = 9406 bytes, creation date = 07/10/2003); (2) Makefile (size = 737 bytes, creation date = 07/10/2003); (3) rtl_crt0.c (size = 7151 bytes, creation date = 07/10/2003); (4) example.c (size = 972 bytes, creation date = 09/24/2003); (5) rtl_mainhelper.h (size = 3847 bytes, creation date = 07/10/2003); and (6) rtl_mainhelper.c (size = 5438 bytes, creation date = 07/10/2003). The contents of the above files are attached hereto (see "source code appendix"). The

computer program listing and the files contained on the compact discs are subject to copyright protection and any use thereof, other than as part of the reproduction of the patent document or the patent disclosure, is strictly prohibited.

BACKGROUND OF THE INVENTION

1. Field of the Invention

[003] The present invention relates, generally, to systems and methods for dynamically linking application software into a running operating system kernel and, more specifically, to systems, software, hardware, products, and processes for use by programmers in creating such application software, and dynamically loading it into and unloading it from a running operating system kernel.

2. Discussion of the Background

[004] There are many software applications in which it is desirable to link the software or one or more modules of the software into a running operating system kernel. For example, it is generally desirable to link device driver software into the operating system kernel because it enables the device driver software to access kernel data space.

[005] Many operating systems offer utilities for dynamically linking application software (a.k.a, "application code" or "application modules") into a running operating system kernel. Typically, these methods involve copying the application module code

and data into kernel memory and then resolving symbol table references. Conventionally, application modules that are to be linked into an operating system kernel include the following sections of code: (1) a section of operating system "kernel" code, (2) a section of code to handle initialization of the module, and (3) a section of code to handle module cleanup when the module is in the process of being removed.

[006] These application modules are considered, essentially, to be dynamically loadable parts of the operating system kernel. But operating system kernels provide a very complex and low level interface to these application software modules, complicating the process of developing such application software and requiring the programmer to directly address and write operating system kernel code, code to handle initialization/loading of the application software, and code to handle unloading/cleanup after termination. Furthermore, there is generally no defined and stable application programming interface within an operating system, and required data structures and interfaces change especially rapidly in operating systems, such as, for example, BSD and Linux, further complicating the programmer's development and implementation of the application software. Creating and loading/unloading the modules is often complex, time consuming, and requires constant monitoring and updating of the code to ensure reliable, error-free implementation.

[007] What is desired, therefore, are systems and methods to overcome the above described and other disadvantages of the conventional system and methods for dynamically linking modules into a running operating system kernel.

SUMMARY OF THE INVENTION

[008] The present invention provides systems and methods for dynamically linking modules into a running operating system kernel. The systems and methods of the present invention overcome the above described and other disadvantages of the conventional systems and methods. For example, the systems and methods specified herein (1) permit an application programmer to write, compile, execute, and terminate application code that is to be loaded into a kernel as if the application code was an ordinary (i.e., not kernel loadable) application program, (2) allow a standard programming environment to be used to encapsulate application software in a familiar environment, and (3) permit automatic cleanup of errors and freeing of program resources when the application terminates. The present invention preserves the advantages of in-kernel programming by providing an application program access to the hardware address space seen by the kernel, as well as access to kernel data structures, internal kernel services and to privileged machine instructions.

[009] Advantageously, the present invention can be applied to a wide range of application

environments (what the application programmer sees) and kernel environments (the underlying programming environment). For example, we have implemented the method for the POSIX threads application environment and the RTLinux and RTCore BSD kernel environments. The RTLinux kernel environment is described in U.S. Patent No. 5,995,745, the contents of which are incorporated herein by reference.

[0010] In one embodiment, the system of the present invention enables a programmer to dynamically link application code created by the programmer into a running operating system kernel , wherein the system includes the following components: (1) an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment; (2) a build system for constructing a loadable module from the application code and the environment library; (3) an execution library comprising one or more routines for encapsulating the loadable module within a standard executable program file, transparently loading the loadable module into the running operating system kernel, setting up input/output channels that may be required, passing arguments to the loadable module, and terminating and unloading the loadable module after receiving a termination signal; (4) an infrastructure library comprising one or more routines that may need to be executed prior to loading the loadable module into the running operating system kernel and/or after unloading the loadable module from the kernel (such

routines may include routines to allocate stable memory (memory that will need to be held after the module completes and is unloaded), routines to initialize a list of modules that will be managed, routines to free memory used by the module, and routines to close files and free semaphores and other resources used by the module); and (5) a build system for constructing the executable program from the loadable module and the execution library, wherein the executable program may be in several files or a single file.

[0011] In another aspect, the present invention includes a computer readable medium, such as, for example, an optical or magnetic data storage device, having stored thereon the execution library, the environment library, the infrastructure library, and the build system.

[0012] The above and other features and advantages of the present invention, as well as the structure and operation of preferred embodiments of the present invention, are described in detail below with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] The accompanying drawings, which are incorporated herein and form part of the specification, illustrate various embodiments of the present invention and, together with the description, further serve to explain the principles of the invention and to enable a person skilled in

the pertinent art to make and use the invention. In the drawings, like reference numbers indicate identical or functionally similar elements. Additionally, the left-most digit(s) of a reference number identifies the drawing in which the reference number first appears.

[0014] FIG. 1 is a functional block diagram that illustrates the components of a system according to one embodiment of the invention.

[0015] FIG. 2 is a diagram illustrating a first function of a build system component of the invention.

[0016] FIG. 3 is a diagram illustrating a second function of a build system component of the invention.

[0017] FIG. 4 is a flow chart illustrating a process 400 that may be performed by build system component of the invention.

[0018] FIG. 5 is a flow chart illustrating a process performed by the "main" routine of the execution library component of the invention.

[0019] FIG. 6 illustrates example pseudo-code of an example loadable module.

[0020] FIG. 7 illustrates a representative computer system for implementing the systems and methods of the present invention for dynamically linking application software into a running operating system kernel.

DETAILED DESCRIPTION OF THE PREFERRED EMBODMENT

[0021] In the following description, for purposes of explanation and not limitation, specific details are set forth, such as particular systems, computers, devices, components, techniques, computer languages, storage techniques, software products and systems, operating systems, interfaces, hardware, etc. in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced in other embodiments that depart from these specific details. Detailed descriptions of well-known systems, computers, devices, components, techniques, computer languages, storage techniques, software products and systems, operating systems, interfaces, and hardware are omitted so as not to obscure the description of the present invention.

[0022] FIG. 1 is a functional block diagram that illustrates the components of a system 100, according to one embodiment of the invention, for dynamically linking application code 102 into a running operating system kernel 104. Application code 102 is code written by a user 101 of system 100. In one embodiment, the code is written as a standard "C" program and includes the "main" function. Example application code that can be used with system 100 is provided in the file "example.c", which is included in the above referenced computer program listing appendix.

[0023] As shown in FIG. 1, system 100 includes an infrastructure library 110, an environment library 112, an execution library 114, and a build system 116. As used herein, the term "library" means: "a set of one or more routines," and the term "routine" means: "a set of one or more program instructions that can be executed." Preferably, each library 112, 114, and 116 is stored in a separate file, but this need not be the case. The libraries could all be stored in one file, in multiple files, or in any other suitable storage architecture or arrangement.

[0024] Build system 116 is configured to construct a "loadable module" 202 from application code 102 and environment library 112, as is illustrated in FIG. 2. As used herein the term "loadable module" means "object code produced from the application code and environment library." In one embodiment, the build system 116 includes one or more makefiles. As further illustrated in FIG. 3, build system 116 is configured to construct an executable program 302 from loadable module 202 and execution library 114.

[0025] FIG. 4 is a flow chart illustrating a process 400 that may be performed by build system 116 to create loadable module 202 and executable program 302. Process 400 can be used when, for example, the C programming language is used in implementing the invention or with any other comparable programming language. Process 400 begins in step 402, where build system 116 compiles application code 102 into object code. Next (step

404), build system 116 links the object code with the environment library 112 object code to produce a linked object module. Next (step 406), build system 116 converts the linked object module into a C code array. This C code array is the "loadable module." Next (step 408), build system 116 compiles the C code array produced in step 406 to produce an object file. Next (step 410), system 116 links the object file produced in step 408 with execution library 112 object code to produce the executable program 302.

[0026] In one embodiment, when user 101 executes program 302, a routine from the execution library 114 sets up an input channel and an output channel by connecting the standard output and standard input of the original process to the standard output and input of the process that will insert the module, and to the fifos that connect user space to kernel space. Setting up the input/output channels is optional if application code 102 does not use input/output channels. Also, when program 302 is executed a routine from execution library 114 inserts loadable module 202 into the operating system address space. Once loadable module 202 is inserted into the operating system address space, loadable module 202 begins to execute.

[0027] If application code 102, which is included in loadable module 202, uses input/output channels, then a routine from execution library 114 waits for loadable module 202 to connect via kernel/user channels, which are implemented as RT-fifos in this embodiment, and then connects those kernel/user

channels to the input/output channels mentioned above.

[0028] After loadable module 202 begins to execute, a routine from environment library 112 creates kernel/user channels, creates a thread to execute application code 102, and then waits for the thread to complete. Creating the kernel/user channels is optional. When the thread completes, a routine from environment library 112 frees resources and unloads loadable module 202. The routines from environment library 112 may need to call routines from infrastructure library 110 for carrying out their work. For example, environment library 112 may delegate some cleanup operations to infrastructure library 110 so that cleanup can take place after loadable module 202 has been unloaded.

EXAMPLE CODE FOR IMPLEMENTING THE INVENTION

[0029] The following examples of code are for the Linux, RTLinux, and/or RTCore BSD operating systems. However, the present invention is not limited to operating only within the Linux operating system. The present invention can, for example, work with other operating systems, including: Net BSD and Free BSD, Apple OS X, other UNIX type operating systems, WindRiver's VxWorks system, and Microsoft's XP and NT operating systems.

[0030] Example code for implementing execution library 114 is provided in the file "user.c", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0031] The example code for implementing execution library 114 is written in the C language and includes a "main" routine. This main routine is the first routine that is executed after user 101 executes executable program 302. FIG. 5 is a flow chart illustrating the process 500 performed by the main routine. As shown in FIG. 5, the main routine collects the arguments to be passed to loadable module 202 (step 502). Next (step 504), the main routine sets up the input and output channels. Next (step 506), main routine creates a child process by executing the fork() routine. The child process immediately replaces its process image with the insmod process image. The child process does this by executing the execl() routine.

[0032] Next (step 508), the main routine pipes loadable module 202 to the insmod process. This causes the insmod process to put loadable module 202 with arguments in kernel address space. When the application code within loadable module 202 begins running in kernel space, a routine within loadable module 202 will create kernel/user channels (e.g., RTCore "fifos") that move data between kernel and user space.

[0033] The main routine waits until it can open these kernel/user channels (step 510). Once it can open the channels it does and then uses the channels to transfer data back and forth so that data coming from loadable module 202 is sent to the standard output of executable program 302 and data coming into the standard input of executable program 302 is

transmitted via the kernel/user channel to loadable module 202 (step 512).

[0034] Example code for implementing environment library 112 is provided in the file "rtl_crt0.c", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0035] As discussed above, environment library 112 includes one or more routines for insulating application code 102 from the operating system environment and implementing a uniform execution environment. That is, instead of depending on a changeable and specialized operating system interface, the program can use standard "C" interfaces - or alternatively, any standard application programming interface. For example, in the current embodiment of the invention, instead of using a Linux kernel "sys_open" operation applied to some terminal device so that the program can output data, the program can simply use the standard "printf" routine. Instead of using some OS dependent routine for generating a thread, the program can use "pthread_create" - a POSIX standard function.

[0036] As shown in the example code, environment library 112 includes an initialization routine called "init_module()" for performing the functions of insulating application code 102 from the operating system environment and implementing a uniform execution environment. The init_module routine is executed as soon as the module is loaded

into the kernel address space and is executed in the context of the process that invokes the insert/load operation. The init_module performs the following steps: (1) copies in arguments that have been passed to it by the execution library and that were passed to the execution library by the user or program that invoked the process, (2) creates the kernel/user channels that connect loadable module 202 to the executable program 302, (3) requests a block of memory from the operating system and stores a "task" structure that describes the application in the module to the infrastructure library, (4) puts the data describing application code 102 on a "task" list that is used by the infrastructure library to control all the modules that it manages, and (5) creates a Linux kernel thread to run a "startup_thread" routine, which is included in the infrastructure library 110 and which is describe below.

[0037] As further shown, environment library 112 may also include a cleanup routine. The cleanup routine included in the example environment library is called "cleanup_module." The cleanup_module routine performs the following task: (1) removes from the task list the task (i.e., the data describing application code 102; (2) waits for the kernel thread to terminate; (3) closes the channels created by the init_module; (4) cleans up state by freeing memory, closing files, and releasing semaphores and possibly other resources; and (5) frees the block of memory that was used to store the task structure.

[0038] Example code for implementing infrastructure library 110 is provided in the files "mainhelper.c" and "mainhelper.h", which are included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0039] The example infrastructure library 110 includes the "startup_thread" routine. As discussed above, the init_module routine creates a Linux kernel thread to run the startup_thread routine. As is illustrated from the example code, the startup_thread routine does some operating system specific setup by detaching the kernel process from any terminals ("daemonize" in Linux), and filling in information in the task structure such as the identity of the current kernel thread, and then executes application code 102 by calling the "task" function. After executing application code 102, the startup_thread routine waits for application code 102 to terminate. When application code 102 terminates, the startup_thread routine sends the application code 102 return value down the output channel, signals completion, and exits. By signaling completion, the startup_thread routine causes the cleanup_module routine to execute.

[0040] The example infrastructure module 110 further includes a routine called "rtl_main_wait." This routine implements the function to wait for the application kernel thread to complete. The function "rtl_main_wait" is called in the application after threads are started so that the "main" routine can

safely be suspended until the subsidiary threads are completed.

[0041] Example code for implementing build system 116 is provided in the file "Makefile", which is included in the above referenced computer program listing appendix. This code is merely an example and should not be used to limit the invention.

[0042] EXAMPLE APPLICATIONS OF THE INVENTION

[0043] In general, all of the modules described below require the addition of application code to a running or booting operating system - the basic functionality provided by loadable kernel modules. Utilizing the present invention, the modules are all (1) simpler to develop and debug, (2) simpler to connect to other components to make a useful software system, and (3) more reliable. Since development time and software reusability (connection to existing software applications) dominate the costs of producing software, the invention provides a significant economic value. In addition, the present invention provides the programmer with a familiar, less complicated development environment, and does not require the programmer to handle complicated kernel-related initialization/loading and unloading/cleanup functions. The invention, thus, facilitates more reliable, error-free application code.

[0044] (1) A real-time data acquisition module under RTLinux:

[0045] The present invention enables a loadable kernel data acquisition module to be developed and tested as if the module were a standard non-kernel module. To develop a real-time data acquisition system in RTLinux using the invention all one needs to do is write a program that runs under the UNIX operating system. The program should include a main program and a thread. The thread should be configured to sample data from the device from which data is to be acquired (e.g., a voltage sensor) at a fixed interval and then write the data to standard output. The program text in pseudo-code 600 is shown in FIG. 6.

[0046] This program can be tested for logical correctness under any POSIX compliant standard UNIX. After testing for logical correctness, the program can be rebuilt using build system 116 to create an executable program that can be run under RTLinux or RTCore BSD, for example. The complexity of launching the thread into the real-time operating system, connecting the thread to data streams, and closing the module on termination is now all hidden to the application. The module can be tested on a UNIX command line with the command: % data_acquisition > test_file.

[0047] (2) A device driver module:

[0048] As noted herein, the invention provides a means of simplifying and automating module deployment and testing. Currently, the most common use for loadable kernel modules is for device drivers that are not statically linked into the

operating system. Drivers in loadable kernel module form allow an operating system to configure itself on boot - adding drivers for devices it detects - and allowing users and system administrators to update systems by, for example, upgrading a disk drive or a communications device and adding a new driver without re-booting a system and interrupting other functions.

[0049] However, driver modules are often very dependent on specific releases of the operating system and are notorious for failing to cooperate with other components. Using the invention, a driver developer can insulate the driver from non-relevant operating system internal changes. More importantly, a system administrator can rely on the automatic cleanup provided by the invention to improve reliability and rely on the automatic setup of communication channels to improve reports. In a Linux system, without the invention, the method for testing a loadable kernel driver module might involve the following steps: (1) the administrator logs in as root to get the correct privileges; (2) the administrator uses the "insmod" utility to attempt to load the driver; and (3) if the insmod utility works, the system administrator uses a "dmesg" utility to print out internal operating system diagnostic messages and to see if he or she can find one relevant to the driver. Suppose that the driver cannot correctly start because, for example, a prior driver for the device has not been removed. At this point, the system administrator must use the "rmmod" tool to try to remove both the

new driver and the prior one, and then again tries to install the new driver. The driver writer must have correctly handled the complex case of responding to an "rmmod" request.

[0050] With the invention, the method for testing the loadable kernel driver is much simpler. The method might include simply the step of running the executable program that loads the module into the kernel. If the module needs to be unloaded from the kernel, the administrator need only execute the kill command to kill the executable program.

[0051] (3) An encryption module:

[0052] This example illustrates the value of the automatic creation of input/output channels by the invention. Suppose that we have a generic operating system module that provides an encryption and security stamp facility and want to attach it to a database. The command line for initiating secure operation of the database in a system utilizing the invention might be:

```
%(decrypt_input | my_database | encrypt_output)&
```

so that the two security modules (i.e., "decrypt_input" and "encrypt_output") are automatically loaded and connected to the inputs and outputs of database module, "my_database". The entire system can be terminated and automatically unloaded with a single signal.

[0053] (4) A security module:

[0054] Loadable kernel modules are themselves potentially a security weakness of an operating

system, since modules traditionally operate within the address space and with all privileges of the operating system itself. There is generally only a check on whether the user loading the module has sufficient privileges to load a module. However, the invention makes it convenient to add more sophisticated security checking either directly in the infrastructure libraries or in a root module that controls all module loads after loading. This module can validate certificates and even provide dynamic code check.

[0055] (5) A fault tolerant module:

[0056] The invention provides a means of dynamically adding a fault tolerance capability to a running operating system by adding a kernel data logger. For example, the script % checkpoint_kernel | netcat 10.0.0.244:45 runs a fault tolerance module named "checkpoint_kernel" and sends the module's output to a standard program (i.e., "netcat") that directs output to a named internet site and TCP port. Using prior methods, the programmer would have had to hand code creation of an output channel in the checkpoint_kernel module and then produce further code to redirect the data to a destination and to process the destination IP and port as parameters.

[0057] FIG. 7 is an illustration of a representative computer system for implementing the systems and methods of the present invention for dynamically linking application software into a running operating system kernel. With reference to

FIG. 7, the method of the present invention may be advantageously implemented using one or more computer programs executing on a computer system 702 having a processor or central processing unit 704, such as, for example, a workstation, server, or embedded-single-board computer using, for example, an Intel-based CPU, such a Centrino, running one of the operating systems previously described, having a memory 706, such as, for example, a hard drive, RAM, ROM, a compact disc, magneto-optical storage device, and/or fixed or removable media, having a one or more user interface devices 708, such as, for example, computer terminals, personal computers, laptop computers, and/or handheld devices, with an input means, such as, for example, a keyboard 710, mouse, pointing device, and/or microphone. The computer program is stored in memory 11 along with other parameters and data necessary to implement the method of the present invention.

[0058] In addition, the computer system 702 may include an analog-to-digital converter, sensors, and various input-output devices, and may be coupled to a computer network, which may also be communicatively coupled to the Internet and/or other computer network to facilitate data transfer and operator control.

[0059] The systems, processes, and components set forth in the present description may be implemented using one or more general purpose computers, microprocessors, or the like programmed according to the teachings of the present specification, as will

be appreciated by those skilled in the relevant art(s). Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the relevant art(s). The present invention thus also includes a computer-based product which may be hosted on a storage medium and include instructions that can be used to program a computer to perform a process in accordance with the present invention. The storage medium can include, but is not limited to, any type of disk including a floppy disk, optical disk, CDROM, magneto-optical disk, ROMs, RAMs, EPROMs, EEPROMs, flash memory, magnetic or optical cards, or any type of media suitable for storing electronic instructions, either locally or remotely.

[0060] While the processes described herein have been illustrated as a series or sequence of steps, the steps need not necessarily be performed in the order described, unless indicated otherwise.

[0061] The foregoing has described the principles, embodiments, and modes of operation of the present invention. However, the invention should not be construed as being limited to the particular embodiments described above, as they should be regarded as being illustrative and not as restrictive. It should be appreciated that variations may be made in those embodiments by those skilled in the art without departing from the scope of the present invention. Obviously, numerous modifications and variations of the present

invention are possible in light of the above teachings. It is therefore to be understood that the invention may be practiced otherwise than as specifically described herein.

[0062] Thus, the breadth and scope of the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

Source Code Appendix:

The following is an example of a loadable kernel module.

```
/* Copyright (C) Finite State Machine Labs Inc., 1995-2003. All rights reserved. */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_t thread;

void *thread_code(void *t) {
     struct timespec next;
     clock_gettime( CLOCK_REALTIME, &next );
     while ( 1 ) {
          timespec_add_ns( &next, 1000*1000 );
clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);
          printf("inside thread\n");
     }
     return NULL;
}

int main(void) {
     int ret;
     ret = pthread_create( &thread, NULL, thread_code, (void *)0 );
     if (ret) {
          printf("Error on create\n");
          return -1;
     }

     /* wait for the thread to exit or an asynchronous signal to stop us */
     rtl_main_wait();

     ret = pthread_cancel( thread );
     if (ret) {
          printf("Error on cancel\n");
          return -2;
     }
     ret = pthread_join( thread, NULL );
     if (ret) {
          printf("Error on join\n");
          return -31;
     }
```

```
        return 0;
    }
```

```
#
# MAKEFILE - __RTLINUX_COPYRIGHT__
#

include ../rtl.mk

CFLAGS := ${SYSCFLAGS}

ifeq ($(CONFIG_UCLINUX),)
all: rtl_crt0.o array user.o user_nostdout.o copy
else
all: array copy
endif

ifeq ($(CONFIG_RTL_BSD),y)
USER_EXTRA += user_elf.o
endif

copy:
	mkdir -p ../libs
	-cp -f rtl_crt0.o user_nostdout.o user.o array
../libs

user.o: user.c $(USER_EXTRA)
	$(CC) $(INCLUDE) -c user.c -o temp.o
	$(LD) -r -o user.o temp.o $(USER_EXTRA)
	@rm -f temp.o

user_nostdout.o: user.c $(USER_EXTRA)
	$(CC) $(INCLUDE) -DRTL_NO_STDOUT=1 -c user.c -o
temp.o
	$(LD) -r -o user_nostdout.o temp.o
$(USER_EXTRA)
	@rm -f temp.o

array: array.c
	$(HOSTCC) -o array array.c

clean:
	rm -f *.o *.rtl array

include ../Rules.make
```

```
/*
 * __RTLINUX_COPYRIGHT__
 */
#include <rtl_conf.h>
#include <rtl_gpos.h>
#include <rtl_printf.h>
#include <rtl_pthread.h>
#include <rtl_sched.h>
#include <rtl_posixio.h>
#include <rtl_mainhelper.h>
#include <rtl_unistd.h>
#include <rtl_fifo.h>
#include <sys/mman.h>
#include <stdio.h>

#ifdef CONFIG_RTL_NETBSD
#include <sys/kthread.h>
#endif

#ifdef CONFIG_RTL_LINUX
int stdout_fifo = -1;

/* per-task info for RTLinux to keep track of
applications */
extern struct per_task_info *info;

extern int main(int argc, char **);

int argc;
char *argv[RTL_MAIN_MAXARGS];
MODULE_PARM(argv, "1-"
__MODULE_STRING(RTL_MAIN_MAXARGS) "s");
MODULE_PARM(argc,"i");

int init_module(void)
{
     struct per_task_info *inf;

     /* setup our task info */
     if ( !(inf = (struct per_task_info *)kmalloc(
                  sizeof(struct per_task_info),
GFP_KERNEL)) )
          return -1;
     inf->module = &__this_module;
     inf->mainfunc = main;
     /*
```

```
        * If we're loaded by the user with insmod
    instead of
        * by the user program with the module built
    into it
5       * then argc/argv may not be set right.  Setup
    some
        * safe defaults if so.  -- Cort
    <cort@fsmlabs.com>
        */
10      if ( !argc ) {
            argc = 1;
            argv[0] = "this_program";
        }
        inf->argc = argc;
15      inf->argv = argv;
        init_completion( &inf->mainwait );
        init_completion( &inf->waitexit );

        /* create the stdout FIFO */
20      {
            int ret;
            char fnam[16];

            inf->stdout_pid = current->pid;
25          sprintf(fnam, "/dev/stdout.%d", inf-
    >stdout_pid);
            if ( (ret = rtl_mkfifo(fnam, 0666)) ) {
                printk("mkfifo() of %s failed\n",
                        fnam);
30              rtl_perror("mkfifo()");
            }

            /* open the file for read only */
            if ( (stdout_fifo = rtl_open(fnam,
35  RTL_O_WRONLY|RTL_O_NONBLOCK)) < 0 ) {
                printk("open() of %s failed\n",
    fnam);
                rtl_perror("open()");
            }
40          inf->stdout_fifo = stdout_fifo;
        }

        /* add this to the list */
        spin_lock( &per_task_lock );
45      inf->next = info;
        info = inf;
        spin_unlock( &per_task_lock );
```

```
        if ( kernel_thread(startup_thread, (void *)inf,
           CLONE_FS | CLONE_FILES | CLONE_SIGNAL |
SIGCHLD ) <= 0 )
            return -1;
    else
            return 0;
}

void cleanup_module(void)
{
        struct per_task_info *inf, *free = NULL;

        /* remove entry this from the list */
        spin_lock(&per_task_lock);
        if (info && (info->module == &__this_module)) {
            free = info;
            info = info->next;
        } else {
            inf = info;
            while (inf) {
                if (inf->next->module ==
&__this_module) {
                    free = inf->next->next;
                    inf = inf->next;
                    break;
                }
            }
        }
        spin_unlock(&per_task_lock);

        if (!free) {
            rtl_printf("Did not find task on
list!\n");
            return;
        }

        /* wakeup the GPOS task */
        complete(&free->mainwait);

        /* wait for the GPOS task to exit */
        wait_for_completion(&free->waitexit);

        /* remove the stdout FIFO */
        {
            char fnam[16];
            int ret;

            rtl_close(stdout_fifo);
```

```
            sprintf(fnam, "/dev/stdout.%d", free-
>stdout_pid);
            if ( (ret = rtl_unlink(fnam)) < 0 ) {
                printk("unlink() of %s failed\n",
fnam);
                rtl_perror("unlink()");
            }
        }

        /* cleanup any state this module left */
        rtl_cleanup_module(free);

        /* free the structure now that it's done */
        kfree(free);
}

#endif /* CONFIG_RTL_LINUX */


#ifdef CONFIG_RTL_NETBSD

int rtl_main_split_args(const char *cmdline, char
**args);
void rtl_main_free_args(char **args);

/* Module management functions */
int rtl_main_init_module(struct per_task_info *inf)
{
        const unsigned int buf_size = 256;
        struct rtl_module *module = inf->module;
        struct proc *pptr = curproc->p_pptr;
        char *cmdline;
        int ret;

        cmdline = malloc(buf_size, M_TEMP, M_WAITOK |
M_ZERO);
        if (cmdline == NULL)
            return -ENOMEM;

        /* modload is called via a shell script */
        inf->stdout_pid = pptr->p_pid;

        /* split up the arguments */
        inf->argv[0] = NULL;
        inf->argc = rtl_main_split_args(inf-
>__rtl_main_args, inf->argv);
        if (inf->argc < 0) {
            ret = -ENOMEM;
```

```
            goto out;
        }

        if (inf->argc == 0) {
            /* no arguments, setup sane defaults */
            snprintf(inf->argv[0], RTL_MAIN_ARGLEN,
"%s", module->name);
            inf->argc = 1;
        }

        /* initialize completion idents */
        init_completion(&inf->mainwait);
        init_completion(&inf->waitexit);

        /* create the stdout FIFO */
        sprintf(cmdline, "/dev/stdout.%d", inf-
>stdout_pid);

        /*
         * attribute this one to the core system,
otherwise our
         * refcount will be high, ditto for open
         */
        if ((ret = __rtl_mkfifo(cmdline, 0666, NULL)))
{
            rtl_printf("mkfifo() of %s failed\n",
cmdline);
            rtl_perror("mkfifo()");
        }

        /* open the file for write only */
        inf->stdout_fifo = __rtl_open(cmdline,
RTL_O_WRONLY|RTL_O_NONBLOCK, NULL);
        if (inf->stdout_fifo  < 0) {
            rtl_printf("open() of %s failed\n",
cmdline);
            rtl_perror("open()");
        }

        /* Add this task to the head of the global list
*/
        simple_lock(&per_task_lock);
        SLIST_INSERT_HEAD(&rtl_task_list, inf, list);
        simple_unlock(&per_task_lock);

        /*
         * finally, fire up our kernel thread, upon
successful exit,
```

```
        * inf->tsk has our task struct and we have a
    kernel thread
        * called module->name.
        */
5
       ret = kthread_create1(startup_thread, inf,
    &inf->tsk, "%s",
                        module->name);

10  out:
       free(cmdline, M_TEMP);
       return ret;
    }

15  int rtl_main_cleanup_module(struct per_task_info
    *inf)
    {
       char buf[32];
       int ret;
20
       simple_lock(&per_task_lock);
       SLIST_REMOVE(&rtl_task_list, inf,
    per_task_info, list);
       simple_unlock(&per_task_lock);
25
       /* wakeup the GPOS task */
       complete(&inf->mainwait);

       /* wait for GPOS task completion */
30     wait_for_completion(&inf->waitexit);

       /*
        * remove our stdout fifo, we don't have to
    lock anymore because
35      * we're not in the rtl_task_list lock anymore
        */
          rtl_close(inf->stdout_fifo);
          sprintf(buf, "/dev/stdout.%d", inf-
    >stdout_pid);
40        if ((ret = rtl_unlink(buf)) < 0 ) {
                  rtl_printf("unlink() of %s
    failed\n", buf);
                  rtl_perror("unlink()");
                  simple_unlock(&per_task_lock);
45                return ret;
          }

       /* cleanup any state the module left behind */
```

```
        rtl_cleanup_module(inf);

        rtl_main_free_args(inf->argv);
        return 0;
}

/*
 * rtl_main_split_args - split arguments in @cmdline
into an argv array
 * @cmdline: the argument list
 * @args: the array to store the returned arguments,
the function
 *      will only allocate the string array.
 */
int rtl_main_split_args(const char *cmdline, char
**args)
{
        const int m_flags = M_WAITOK | M_ZERO;
        int i, j, idx;
        char *buf;

        buf = malloc(RTL_MAIN_MAXARGS *
RTL_MAIN_ARGLEN, M_TEMP, m_flags);
        if (!buf)
                return -ENOMEM;

        for (i = 0; i < RTL_MAIN_MAXARGS; i++)
                args[i] = buf + (i * RTL_MAIN_ARGLEN);

        i = 0, j = 0, idx = 0;
        while (cmdline[i] && (idx < RTL_MAIN_MAXARGS))
{
                if (j >= RTL_MAIN_ARGLEN)
                        break;

                args[idx][j] = cmdline[i];

                if (isspace(cmdline[i])) {
                        args[idx][j] = '\0';
                        idx++; j = 0;
                } else
                        j++;
                i++;

        }
        return idx;
}
```

```
void rtl_main_free_args(char **args)
{
     free(args[0], M_TEMP);
}

#endif     /* CONFIG_RTL_NETBSD */
```

```
/*
 * __RTLINUX_COPYRIGHT__
 */
#include <rtl_gpos.h>
#include <rtl_mainhelper.h>
#include <rtl_printf.h>
#include <rtl_sched.h>
#include <rtl_posixio.h>
#include <rtl_unistd.h>
#include <posix/sys/mman.h>
#include <posix/unistd.h>

#ifdef CONFIG_RTL_NETBSD
#include <sys/signal.h>
#include <sys/kthread.h>

struct simplelock per_task_lock =
SIMPLELOCK_INITIALIZER;
struct rtl_task_list_s rtl_task_list =
SLIST_HEAD_INITIALIZER(rtl_task_list);
#endif /* CONFIG_RTL_NETBSD */


#ifdef CONFIG_RTL_LINUX
/* per-task info for RTLinux to keep track of
applications */
spinlock_t per_task_lock = SPIN_LOCK_UNLOCKED;
struct per_task_info *info = NULL;
#endif /* CONFIG_RTL_LINUX */

kthread_ret_t startup_thread(void *arg)
{
     struct per_task_info *inf = (struct
per_task_info *)arg;
     int ret, temp = -1;

#ifdef CONFIG_RTL_LINUX
     /* detach this thread from the user/tty */
     daemonize();

     /*
      * update some per-process information, inf-
>tsk is already
      * set to curproc on NetBSD
      */
     inf->tsk = current;
#endif /* CONFIG_RTL_LINUX */
```

```
        inf->thread = inf->module;

        /* call the task main routine */
        ret =  inf->mainfunc(inf->argc, inf->argv);

        /* we have the return value, send it to the
    user-program */
        rtl_write(inf->stdout_fifo, &temp,
    sizeof(temp));
        rtl_write(inf->stdout_fifo, &ret, sizeof(ret));

        /* wake up the cleanup_module() routine if it's
    waiting on us */
        complete(&inf->waitexit);

        /*
         * We always return success here.  That way,
    the module
         * loading always appears successful for the
    user-program
         * unless there was an actual load problem
    (unresolved references,
         * bad file and so on).
         *
         * When there is an error from main() or any
    other operation
         * inside the application then we return the
    error value above
         * through the stdout FIFO.
         *     -- Cort <cort@fsmlabs.com>
         */
    #ifdef CONFIG_RTL_LINUX
        return 0;
    #endif

    #ifdef CONFIG_RTL_NETBSD
        kthread_exit(0);
    #endif
    }

    void __rtl_main_wait(struct rtl_module *module)
    {
        struct per_task_info *inf = NULL;

    #ifdef CONFIG_RTL_LINUX
        /* remove entry this from the list */
        spin_lock(&per_task_lock);
        inf = info;
```

```
        while (inf) {
            if (inf->module == module)
                break;
            inf = inf->next;
        }
        spin_unlock(&per_task_lock);
#endif /* CONFIG_RTL_LINUX */

#ifdef CONFIG_RTL_NETBSD
        simple_lock(&per_task_lock);
        SLIST_FOREACH(inf, &rtl_task_list, list) {
            if (inf->module == module)
                break;
        }
        simple_unlock(&per_task_lock);
#endif /* CONFIG_RTL_NETBSD */

        if (!inf) {
            rtl_printf("%s:%d !inf\n", __FILE__,
__LINE__);
            return;
        }

        /* wait for an rmmod event */
        wait_for_completion(&inf->mainwait);
}

#ifdef CONFIG_RTL_NETBSD
/* sleep until completion wakeup, the ->done is so
that if a thread
 * completes before the waiter the waiter can
proceed
 */
void wait_for_completion(struct completion *ident)
{
        simple_lock(&ident->lock);
        while (ident->done == 0) {
            ltsleep(ident, curproc->p_priority |
PCATCH,
                __FUNCTION__, 0, &ident->lock);
        }
        simple_unlock(&ident->lock);
}

void complete(struct completion *ident)
{
        simple_lock(&ident->lock);
        ident->done++;
```

```
        simple_unlock(&ident->lock);
        wakeup(ident);
    }
    #endif

    /*
     * Cleanup after a rtlinux module that has exited
     */
    void rtl_cleanup_module(struct per_task_info *inf)
    {
        int i;

    #if 0
        rtl_pthread_t t;
        /* cancel/join all threads for this module */
    repeat:
        rtl_spin_lock( &sched_data(i)->rtl_tasks_lock
    );
        for ( i = 0; i < rtl_num_cpus(); i++ ) {
            t = sched_data(i)->rtl_tasks;
            while( t != NULL ) {
                if ( t->creator == &__this_module ) {
                    rtl_printf("Canceling thread
    %08x\n", t);
                    rtl_spin_unlock( &sched_data(i)-
    >rtl_tasks_lock );
                    if ( rtl_pthread_cancel( t ) )
                        rtl_printf("cancel
    failed\n");
                    if ( rtl_pthread_join( t, NULL )
    )
                        rtl_printf("join
    failed\n");
                    goto repeat;
                }
                t = t->next;
            }
            rtl_spin_unlock( &sched_data(i)-
    >rtl_tasks_lock );
        }
    #endif

        /* kill any FDs that this task opened */
        for ( i = 0 ; i < CONFIG_RTL_MAX_FILES; i++ ) {
            incr_fd_usage(i);
            if ( (rtl_fds[i].f_op) &&
                 (rtl_fds[i].creator == inf->module) )
                 rtl_close( i );
```

```
            decr_fd_usage(i);
        }

        /* unregister any devices that this task
 5  managed */
        for ( i = 0 ; i < CONFIG_RTL_MAX_DEV; i++ ) {
            int j;

            incr_dev_usage(i);
10          if ( (rtl_inodes[i].valid) &&
                 (rtl_inodes[i].creator == inf-
    >thread) ) {
                 /* find all FDs that have this file
    open */
15               for ( j = 0 ; j <
    CONFIG_RTL_MAX_FILES; j++ ) {
                     incr_fd_usage(j);
                     if ( (rtl_fds[j].f_op) &&
                          (rtl_fds[j].devs_index ==
20  i) )
                          rtl_close( j );
                     decr_fd_usage(j);
                 }
                 if (i >= CONFIG_RTL_MAX_LEGACY_DEV) {
25                   /* need to unlink
    unconditionally */
                     if (rtl_inodes[i].has_gpos_file)
    {

30      rtl_gpos_unlink(rtl_inodes[i].name);
                          rtl_inodes[i].has_gpos_file
    = 0;
                     }

35                   /* The user may have pushed
    usage to any
                     amount. Reset and handle now. */

        rtl_xchg(&rtl_inodes[i].use_count,1);
40
                     /* The GPOS user may still be
    out there.
                     Let them go - they'll clean up.
    Otherwise,
45                   do it here. (the decr will drop
    usage 1 to 0.  */
                     if (!rtl_gpos_devices[i].active)
    {
```

```
                                decr_dev_usage(i);

            rtl_gpos_unregister_dev(rtl_inodes[i].name);

            rtl_unregister_dev(rtl_inodes[i].name);
                            }
                        }
                    } else {
                        decr_dev_usage(i);
                    }
            }

    }
```

```
/*
 * __RTLINUX_COPYRIGHT__
 */

#ifndef __RTL_MAINHELPER__
#define __RTL_MAINHELPER__

#include <rtl_module.h>

#ifndef CONFIG_UCLINUX

/* max # of arguments that can be passed into a
main()
 * program via argc/argv */

#define RTL_MAIN_MAXARGS 5
#define RTL_MAIN_ARGLEN       64

#ifdef __KERNEL__
#ifdef CONFIG_RTL_LINUX

extern spinlock_t  per_task_lock;
struct per_task_info {
     struct rtl_module *module;
     struct rtl_module *thread;
     struct per_task_info *next;
     struct task_struct *tsk;
     int (*mainfunc)(int, char **);
     struct completion mainwait, waitexit;
     int argc;
     char **argv;
     int stdout_fifo, stdout_pid;
};

typedef int kthread_ret_t;
#endif /* CONFIG_RTL_LINUX */

#ifdef CONFIG_RTL_NETBSD

#include <sys/queue.h>

struct completion {
     struct simplelock lock;
     unsigned int  done;
};
```

```
/* Used to protect rtl_task_list
traversal/insertion/removal */
extern struct simplelock per_task_lock;

struct per_task_info {
     struct rtl_module *module;
     struct rtl_module *thread;
     SLIST_ENTRY(per_task_info) list;
     struct proc *tsk;
     int (*mainfunc) (int, char **);
     struct completion mainwait, waitexit;
     int argc;
     char *argv[RTL_MAIN_MAXARGS];
     char *__rtl_main_args;
     int stdout_fifo, stdout_pid;
};

typedef void kthread_ret_t;
SLIST_HEAD(rtl_task_list_s, per_task_info);
extern struct rtl_task_list_s rtl_task_list;
extern void complete(struct completion *);
extern void wait_for_completion(struct completion
*);
#define init_completion(x)    do { \
     simple_lock_init(&(x)->lock); \
     (x)->done = 0; \
} while (0)

/*************************************************
***
 *
 * We only include the following in 'main'
applications
 *

*************************************************
**/

#ifdef KMOD

#undef RTL_MODULE
#undef RTLINUX_MODULE

/*
 * Note the declaration of __main,' there are too
many users of int main(void)
 * and this only results in a warning, however
really they should not be using
```

```
 * that kind of a declaration for their main
function. The correct definition
 * being int main(int, char **) and parameter
passing might have undefined side
 * effects on some systems -Zwane
 */

#define RTLINUX_MODULE(modname) MOD_MISC(#modname) \
static char
__rtl_main_args[RTL_MAIN_MAXARGS*RTL_MAIN_ARGLEN] \

__attribute__((section(".rtl.main.args"))); \
static struct rtl_module __this_module = {.usecount
= 0, .name = #modname}; \
extern int rtl_main_init_module(struct per_task_info
*); \
extern int rtl_main_cleanup_module(struct
per_task_info *); \
static int __main() ;\
static struct per_task_info __inf = { \
     .mainfunc = __main, \
     .module   = &__this_module, \
     .__rtl_main_args = __rtl_main_args \
}; \
static int modname##_module_handle(struct lkm_table
*lkmtp, int    cmd) \
{\
     int err = 0;\
\
     switch (cmd) {\
     case LKM_E_LOAD:\
\
          if (lkmexists(lkmtp))\
               return -EEXIST;\
\
          err = rtl_main_init_module(&__inf);\
          break;\
\
     case LKM_E_UNLOAD:\
\
          if (__this_module.usecount > 0) { \
               return -EBUSY; \
          } \
          err = rtl_main_cleanup_module(&__inf);\
          break;\
\
     default:\
          err = -EINVAL;\
```

```
            break;\
        }\
    \
        return err;\
}\
\
int modname##_lkmentry(struct lkm_table *lkmtp, int
cmd, int ver)\
{\
    DISPATCH(lkmtp,cmd,ver,modname##_module_handle,
\
            modname##_module_handle, lkm_nofunc)\
}

#define RTL_MODULE(modname) RTLINUX_MODULE(modname)

/* This is defined during build to evaluate to
RTL_MODULE(module) */
KMOD

/* The real main entry point */
#define main        static __main
#define stdout_fifo (__inf.stdout_fifo)

#endif                      /* KMOD */
#endif                      /* CONFIG_RTL_NETBSD */

extern void rtl_cleanup_module(struct per_task_info
*);
extern kthread_ret_t startup_thread(void *);
#endif /* __KERNEL__ */
#endif /* CONFIG_UCLINUX */
#endif /* __RTL_MAINHELPER__ */
```

```
/*
 * __RTLINUX_COPYRIGHT__
 */

#include <stdio.h>
#include <stdarg.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/select.h>
#include <sys/wait.h>
#include <sys/utsname.h>
#include <termios.h>
#include <rtl_conf.h>
#include <rtl_mainhelper.h>

char filename[255];
char *paths[] = { "/opt/rtldk-2.0/bin/",
"/opt/rtldk-1.2/bin/",
            "/sbin/", "/bin", "/usr/bin/", NULL };
char *program = NULL;
void setup_signals(void);
void do_printf_read(void);

/* stdout FD from the module */
int f;
/* return value from the module */
int retval = 0, have_retval = 0;

#ifdef CONFIG_RTL_NETBSD
int loading_rtcore = 0;
#endif

int platform_exec(char *cmd, char *module_argc, char
*args)
{
#if defined(CONFIG_RTL_LINUX)
     return execl(cmd, cmd, filename, "-o", program,
                  module_argc, args, NULL);
#elif defined(CONFIG_RTL_NETBSD)
     return execl(cmd, cmd, filename, NULL);
#endif
}
```

```
void prepare_arguments(int argc, char **argv, char
*__argc, char *__argv)
{
     int i;

     __argv[0] = 0;
#if defined(CONFIG_RTL_LINUX)
     /* setup args for the module itself */
     sprintf(__argc, "argc=%d", argc);
     sprintf(__argv, "argv=\"%s\"", argv[0]);
     for ( i = 1; i < argc; i++ )
          sprintf(__argv, "%s,\"%s\"", __argv,
argv[i]);
#elif defined(CONFIG_RTL_NETBSD)
     /* we take the arguments as-is under BSD */
     sprintf(__argv, "%s ", program);   /* argv[0]
*/
     for (i = 1; i < argc; i++) {
          strcat(__argv, argv[i]);
          strcat(__argv, " ");
     }
#endif
}

void handler(int signal)
{
     char command[256];
     int i;
     struct stat buf;

     /* find the right path */
     for ( i = 0 ; paths[i] != 0; i++ ) {
          sprintf(command, "%srmmod", paths[i]);
          if ( !stat( command, &buf ) )
               break;
     }

     /* error if we could not find rmmod */
     if ( !paths[i] ) {
          fprintf(stderr, "Could not find rmmod\n");
          exit(-1);
     }

     sprintf(command, "%s %s", command, program);
     i = system(command);

     if ( i != 0 ) {
```

```
            fprintf(stderr,"%s: received signal but
    removal of module failed.\n", program);
            fprintf(stderr,"There may be other
    programs running that require this module.\n");

            setup_signals();
            return;
        }

    #ifndef RTL_NO_STDOUT
        /* if we don't have the retval set, get one */
        while ( !have_retval ) {
            do_printf_read();
        }
    #endif /* RTL_NO_STDOUT */

        close(f);
        exit(retval);
    }

    void setup_signals(void)
    {
        struct sigaction act;

        /* setup to catch any signals that this process
    receives */
        act.sa_handler = handler;
        act.sa_flags = SA_RESETHAND;
        sigaction(SIGINT, &act, NULL);
        sigaction(SIGQUIT, &act, NULL);
        sigaction(SIGTERM, &act, NULL);
    }

    char cmd[256], args[256], module_argc[16];

    #ifndef RTL_NO_STDOUT
    void do_printf_read(void)
    {
        /* for reading stdout data from the module */
        char *fmt = NULL, *string = NULL;
        int fmtlen = 0;
        int argtype, argsize;
        int i;

        /* read the size of the fmt string */
        if ( read(f, &i, sizeof(i)) <= 0 ) {
            fprintf(stderr, "%s: read of string length
    failed\n", program);
```

```
            perror("read()");
            exit(-1);
        }

        /*
         * special case for when the main() routine in
    the kernel exits on
         * its own
         */
        if ( i == -1 ) {
            /* read the return value */
            if ( read(f, &retval, sizeof(retval)) <= 0
    ) {
                fprintf(stderr, "%s: read of
    retval\n",
                    program);
                perror("read()");
                exit(-1);
            }
            have_retval = 1;
            return;
        }

        /* read the size of the arg */
        if ( read(f, &argsize, sizeof(argsize)) <= 0 )
    {
            fprintf(stderr, "%s: read of arg length
    failed\n", program);
            perror("read()");
            exit(-1);
        }

        /* get the arg type */
        if ( (read(f, &argtype, sizeof(argtype))) <
    sizeof(argtype) ) {
            fprintf(stderr, "%s: read of arg type
    failed\n", program);
            perror("read()");
            exit(-1);
        }

        /* if the string is too large, get more memory
    */
        if ( i+1 > fmtlen ) {
            if ( fmt )
                free(fmt);
            fmtlen = i+1;
```

```
            if ( (fmt = (char *)malloc(fmtlen)) <= 0 )
    {
                fprintf(stderr, "Cannot allocate
    memory for larger format string\n");
                perror("malloc()");
            }
        }

        /* read the format string */
        if ( (read(f, fmt, i)) < i ) {
            fprintf(stderr, "%s: read of fmt string
    failed\n", program);
            perror("read()");
            exit(-1);
        }
        fmt[i] = 0;

        /* if there is no arg, just print the string
    and continue */
        if ( argsize == 0 ) {
            printf(fmt);
            fflush(stdout);
            return;
        }

        /* read the argument */
        switch( (char)argtype ) {
            int d;
            char *s;
            unsigned long l;
            double dl;
        case 'f':
            if ( (read(f, &dl, argsize)) < argsize ) {
                fprintf(stderr, "%s: read of 'f' type
    failed\n", program);
                perror("read()");
                exit(-1);
            }
            printf(fmt, dl);
            fflush(stdout);
            break;
        case 'p':
            if ( (read(f, &l, argsize)) < argsize ) {
                fprintf(stderr, "%s: read of 'p' type
    failed\n", program);
                perror("read()");
                exit(-1);
            }
```

```
            printf(fmt, l);
            fflush(stdout);
            break;
        case 'd':
        case 'i':
        case 'o':
        case 'u':
        case 'x':
        case 'X':
            if ( (read(f, &d, argsize)) < argsize ) {
                fprintf(stderr, "%s: read of 'd' type
    failed\n", program);
                perror("read()");
                exit(-1);
            }
            printf(fmt, d);
            fflush(stdout);
            break;
        case 'c':
            if ( (read(f, &d, argsize)) < argsize ) {
                fprintf(stderr, "%s: read of 'c' type
    failed\n", program);
                perror("read()");
                exit(-1);
            }
            printf(fmt, (char)d);
            fflush(stdout);
            break;
        case 's':
            /* get space for the string */
            if ( (string = (char *)malloc(argsize+1))
    <= 0 ) {
                fprintf(stderr,
                    "Cannot allocate memory for
    string arg\n");
                perror("malloc()");
            }

            if ( (read(f, string, argsize)) < argsize
    ) {
                fprintf(stderr, "%s: read of 's' type
    failed\n", program);
                perror("read()");
                exit(-1);
            }
            string[argsize] = 0;
            printf( fmt, string );
            fflush(stdout);
```

```
                free( string );
                break;
        }
    }
#endif /* RTL_NO_STDOUT */

int main(int argc, char **argv)
{
        extern char ___module_data[];
        extern unsigned long ___module_size;
        int sysret, child, status, i;
        struct stat buf;
        fd_set readfds, writefds, exceptfds;
        mode_t old_umask;
        char *tmp_dir;

        /*
         * strip off the extension, Linux doesn't care
but
         * BSD modload(8) uses it to determine the
module
         * entry point
         */
        program = (char *)basename(argv[0]);
        strtok(program, ".");

        setup_signals();

#ifdef CONFIG_RTL_NETBSD
          /* check whether we're loading rtcore */
          if (!strncmp(program, "rtcore",6)) {
                  loading_rtcore = 1;
                  program = "rtl";
          }
#endif

        tmp_dir = (char *)getenv("TMP");
        sprintf(filename, "%s/%s.o", tmp_dir ? :
"/tmp", program);

        if ( argc > RTL_MAIN_MAXARGS+1 ) {
              fprintf(stderr,"Cannot pass more than %d
arguments\n",
                    RTL_MAIN_MAXARGS);
              return -1;
        }

        old_umask = umask(0600);
```

```
        if ((f = open(filename, O_CREAT | O_WRONLY |
    O_TRUNC)) < 0) {
             perror("open()");
             return -1;
        }
        umask(old_umask);

        prepare_arguments(argc, argv, module_argc,
    args);
    #ifdef CONFIG_RTL_NETBSD
        copy_module_args(___module_data, args);
    #endif

        /* write the module data to the file */
        if ( write(f, ___module_data, ___module_size)
    != ___module_size ) {
             perror("write()");
             close(f);
             unlink(filename);
             return -1;
        }

        close(f);

        /* find the right path */
        for ( i = 0 ; paths[i] != 0; i++ ) {
             sprintf(cmd, "%sinsmod", paths[i]);
             if ( !stat( cmd, &buf ) )
                  break;
        }

        /* error if we could not find insmod */
        if ( !paths[i] ) {
             fprintf(stderr, "Could not find
    insmod\n");
             unlink(filename);
             return -1;
        }

        /* We do a fork and exec here so we get the PID
    of the insmod process
         * so we are able to open the proper
    /dev/stdout.* file.  Also, we
         * need to be careful since system/setuid and
    certain versions of bash
         * do not cooporate well. -- Cort
    <cort@fsmlabs.com>
         */
```

```
        if ( (child = fork()) > 0 ) {
             /* parent */
        } else {
             /* child */

             /* exec the insmod */
             platform_exec(cmd, module_argc, args);

             /* we got here, things aren't looking good
*/
             perror("execl");
             exit(1);
        }

        /* wait for the module load to be done, remove
the file */
        wait(&status);
        unlink(filename);

        if (WEXITSTATUS(status) == 0) {
#ifndef RTL_NO_STDOUT
             /* Try to open the stdout. We don't check
for errors since
              * some modules (rtcore, ckit and so on)
will not produce a stdout
              * FIFO -- Cort <cort@fsmlabs.com>
              */
             sprintf(cmd, "/dev/stdout.%d", child);
             if ( (f = open(cmd, O_RDONLY)) < 0 ) {
                  fprintf(stderr,"%s: unable to open
stdout FIFO\n", cmd);
                  perror("open()");
             }

             /* setup select arguments */
             if ( f >= 0 ) {
                  FD_ZERO(&readfds);
                  FD_ZERO(&writefds);
                  FD_ZERO(&exceptfds);
                  FD_SET(f, &readfds);
             }
#else
             f = -1;
#endif /* RTL_NO_STDOUT */

             while( 1 ) {
#ifndef RTL_NO_STDOUT
                  if ( f >= 0 ) {
```

```
                    select( f+1, &readfds,
&writefds, &exceptfds, NULL );
                    /* loop through here, printing
when necessary */
                    do_printf_read();
                    /* if the last read gave us a
retval, we're done */
                    if ( have_retval )
                         handler(0);
               }
               else
#endif /* RTL_NO_STDOUT */
                    sleep(300);

          }
     } else {
          /* the 'insmod' failed for some reason */
          fprintf(stderr,"%s: Problem loading
module\n", program);
          exit(-1);
     }

     handler(0);

     return 0;
}
```

What is claimed is:

1. A system for dynamically linking application code created by a programmer into a running operating system kernel, comprising:

an environment library comprising one or more routines for insulating the application code from the operating system environment and for implementing a uniform execution environment; and

a build system for constructing a loadable module from the application code and the environment library and for constructing a standard executable program from the loadable module and an execution library, wherein

the execution library comprises one or more routines for transparently loading the loadable module into the running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module after receiving a termination signal.

2. The system of claim 1, further comprising an infrastructure library comprising one or more routines executed prior to loading the loadable module into the running operating system kernel and/or after unloading the loadable module from the kernel.

3. The system of claim 1, wherein the execution library includes one or more routines for setting up

input/output channels.

4. The system of claim 1, wherein the standard executable program may be in several files or a single file.

5. The system of claim 1, wherein one of the one or more routines of the execution library includes code for executing a utility for installing the loadable module into the running operating system kernel.

6. The system of claim 5, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

7. The system of claim 1, wherein the build system includes instructions for compiling the application code into object code.

8. The system of claim 7, wherein the build system further includes instructions for linking said object code with object code from the environment library to produce a linked object module.

9. The system of claim 8, wherein the build system further includes instructions for converting the linked object module into a C code array.

10. The system of claim 8, wherein the build system further includes instructions for compiling

the C code array to produce an object file and for linking said object file with object code from the execution library to produce the standard executable program.

11. The system of claim 1, wherein the environment library includes one or more routines to create kernel/user channels.

12. The system of claim 1, wherein the environment library includes one or more routines to create a thread to execute the application code.

13. The system of claim 12, wherein the environment library includes one or more routines for freeing resources and unloading the loadable module when the thread completes.

14. The system of claim 1, wherein the environment library includes one or more routines for (a) copying in arguments; (b) creating communication channels that connect the loadable module to the executable program; (c) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (d) putting data describing the application code on a task list.

15. The system of claim 14, wherein the environment library further includes one or more routines for (a) removing said data describing the application code from the task list; (b) closing

said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

16. A method, comprising:

creating a loadable module;

creating an executable program; and

executing the executable program, wherein the executable program performs a method comprising the steps of:

setting up input/output channels;

inserting the loadable module into an operating system address space, wherein, once the loadable the module is inserted into the operating system address space, the loadable module begins to execute; and

waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels.

17. The method of claim 16, wherein after the loadable module is inserted into the operating system address space the loadable module performs a method comprising the steps of:

creating kernel/user channels;

creating a thread to execute application code; and

waiting for the thread to complete.

18. The method of claim 17, wherein the method performed by the loadable module further

includes the step of freeing resources after the thread completes.

19. A computer readable medium having computer instructions stored thereon, the computer instructions comprising:

a first set of computer instructions for insulating application code from an operating system environment;

a second set of computer instructions for constructing a loadable module from the application code and the first set of computer instructions; and

a third set of computer instructions for constructing an executable program from the loadable module and a fourth set of computer instructions; wherein

the fourth set of computer instructions includes computer instructions for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal.

20. The computer readable medium of claim 19, wherein the computer instructions for loading the loadable module into the running operating system kernel include computer instructions for executing a utility for installing the loadable module into the running operating system kernel.

21. The computer readable medium of claim 20, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

22. The computer readable medium of claim 19, wherein the second set of computer instructions includes instructions for compiling the application code into object code.

23. The computer readable medium of claim 22, wherein the second set of computer instructions further includes instructions for linking said object code with object code from the environment library to produce a linked object module.

24. The computer readable medium of claim 23, wherein the third set of computer instructions includes instructions for converting the linked object module into a C code array.

25. The computer readable medium of claim 24, wherein the third set computer instructions of further includes instructions for compiling the C code array to produce an object file and for linking said object file with object code from a library to produce the executable program.

26. The computer readable medium of claim 19, wherein the first set of computer instructions

includes instructions for creating kernel/user channels.

27. The computer readable medium of claim 19, wherein the first set of computer instructions includes instructions for creating a thread to execute the application code.

28. The computer readable medium of claim 27, wherein the first set of computer instructions includes instructions for freeing resources and unloading the loadable module when the thread completes.

29. The computer readable medium of claim 19, wherein the first set of computer instructions includes instructions for (a) creating communication channels that connect the loadable module to the executable program; (b) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (c) putting data describing the application code on a task list.

30. The computer readable medium of claim 29, wherein the first set of computer instructions further includes instructions for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

31. The computer readable medium of claim 19, wherein the executable program may be in several files or a single file.

32. A computer system, comprising:
first means for insulating application code from an operating system environment;
second means for constructing a loadable module from the application code and the first means;
third means for constructing an executable program from the loadable module; and
fourth means for transparently loading the loadable module into a running operating system kernel, passing arguments to the loadable module, and terminating and unloading the loadable module from the running operating system kernel after receiving a termination signal.

33. The computer system of claim 32, wherein means for loading the loadable module into the running operating system kernel include means for executing a utility for installing the loadable module into the running operating system kernel.

34. The computer system of claim 33, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

35. The computer system of claim 32, wherein the second means includes means for compiling the application code into object code.

36. The computer system of claim 35, wherein the second means further includes means for linking said object code with object code from the environment library to produce a linked object module.

37. The computer system of claim 36, wherein the third means includes means for converting the linked object module into a C code array.

38. The computer system of claim 37, wherein the third means further includes instructions for compiling the C code array to produce an object file and for linking said object file with object code from a library to produce the executable program.

39. The computer system of claim 32, wherein the first means includes means for creating kernel/user channels.

40. The computer system of claim 32, wherein the first means includes means for creating a thread to execute the application code.

41. The computer system of claim 40, wherein the first means includes means for freeing resources and unloading the loadable module when the thread completes.

42. The computer system of claim 32, wherein the first means includes means for (a) creating communication channels that connect the loadable module to the executable program; (b) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (c) putting data describing the application code on a task list.

43. The computer system of claim 42, wherein the first means further includes means for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

44. The computer system of claim 32, wherein the executable program may be in several files or a single file.

45. A computer system for dynamically linking application code created by a programmer into a running operating system kernel, comprising:

means for creating a loadable module; and

means for creating an executable program that is configured to performs a method comprising the steps of:

setting up input/output channels;

inserting the loadable module into address space of the running operating system kernel,

wherein, once the loadable the module is inserted into the address space, the loadable module begins to execute; and

waiting for the loadable module to connect via kernel/user channels and then connecting those kernel/user channels to the input/output channels.

46. The computer system of claim 45, wherein the loadable module is configured to perform a method after the loadable module is inserted into the operating system address space, wherein said method comprises the steps of:

creating kernel/user channels;

creating a thread to execute the application code; and

waiting for the thread to complete.

47. The computer system of claim 46, wherein the method performed by the loadable module further includes the step of freeing resources after the thread completes.

48. The computer system of claim 45, wherein the step of inserting the loadable module into an operating system address space includes the step of creating a child process, wherein the child process replaces its image with the insmod process image.

49. The computer system of claim 48, wherein the step of inserting the loadable module into an operating system address space further includes the

step of piping the loadable module to the insmod process.

50. A method for dynamically linking application code created by a user into a running operating system kernel, comprising:

constructing a loadable module from application source code written by a user;

creating an executable program, wherein the executable program is configured to transparently load the loadable module into the running operating system kernel;

executing the executable program, thereby loading the loadable module into the running operating system kernel; and

unloading the loadable module from the running operating system kernel by sending a termination signal to the executable program.

51. The method of claim 50, wherein the application source code is an ordinary application program.

52. The method of claim 50, wherein the step of constructing the loadable module from the application source code consists essentially of executing a pre-defined makefile.

53. The method of claim 50, further comprising the step of providing a makefile to the user, wherein the user performs the step of constructing

the loadable module by executing the makefile after the user has created the application code.

54. The method of claim 50, further comprising the step of providing the user with a library comprising object code, wherein the step of constructing the loadable module from the application source code comprises the steps of compiling the application source code into object code; linking the object code with object code from the library to produce a linked object module; and converting the linked object module into a C code array.

55. The method of claim 54, wherein the step of constructing the loadable module further comprises the step of compiling the C code array to produce an object file.

56. The method of claim 55, further comprising the step of providing the user with a second library comprising object code, wherein the step of constructing the executable program comprises the steps of linking the object file with object code from the second library.

57. The method of claim 54, wherein the library includes one or more routines to create kernel/user channels.

58. The method of claim 54, wherein the library includes one or more routines to create a thread to execute the application code.

59. The method of claim 58, wherein the library includes one or more routines for freeing resources and unloading the loadable module when the thread completes.

60. The method of claim 54, wherein the library includes one or more routines for (a) copying in arguments; (b) creating communication channels that connect the loadable module to the executable program; (c) requesting a block of memory from the operating system and storing a structure therein that describes the application code; and (d) putting data describing the application code on a task list.

61. The method of claim 60, wherein the environment library further includes one or more routines for (a) removing said data describing the application code from the task list; (b) closing said communication channels that connect the loadable module to the executable program; and (c) freeing the block of memory that was requested from the operating system.

62. The method of claim 50, wherein the executable program is configured to set up input/output channels.

63. The method of claim 50, wherein the executable program is configured to execute a utility for installing the loadable module into the running operating system kernel.

64. The method of claim 63, wherein the utility for installing the loadable module into the running operating system kernel is the insmod program.

65. The method of claim 63, wherein the step of inserting the loadable module into an operating system address space includes the step of creating a child process, wherein the child process replaces its image with the insmod process image.

66. The method of claim 65, wherein the step of inserting the loadable module into an operating system address space further includes the step of piping the loadable module to the insmod process.

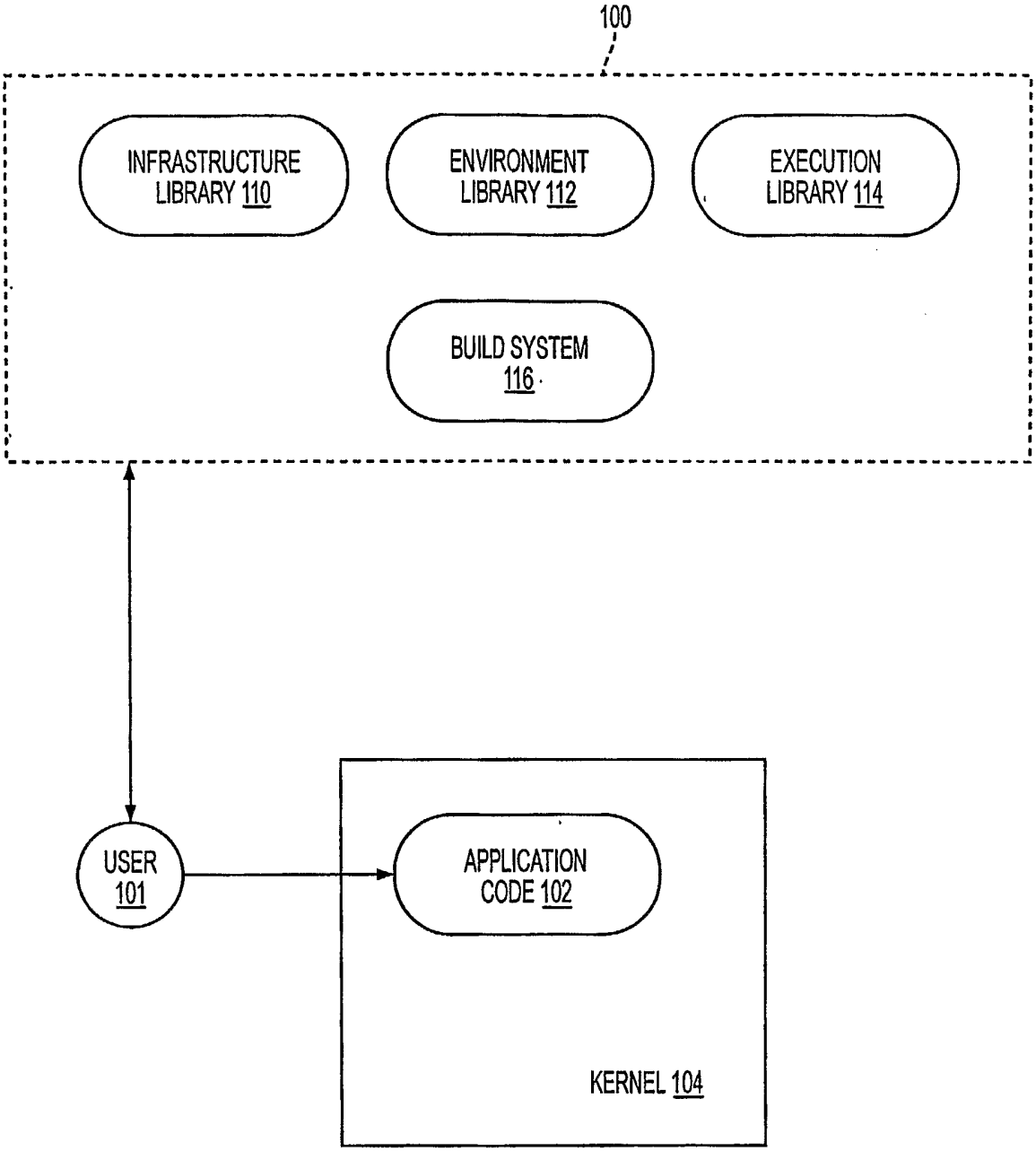

100
INFRASTRUCTURE LIBRARY 110
ENVIRONMENT LIBRARY 112
EXECUTION LIBRARY 114
BUILD SYSTEM 116
USER 101
APPLICATION CODE 102
KERNEL 104

FIG. 1

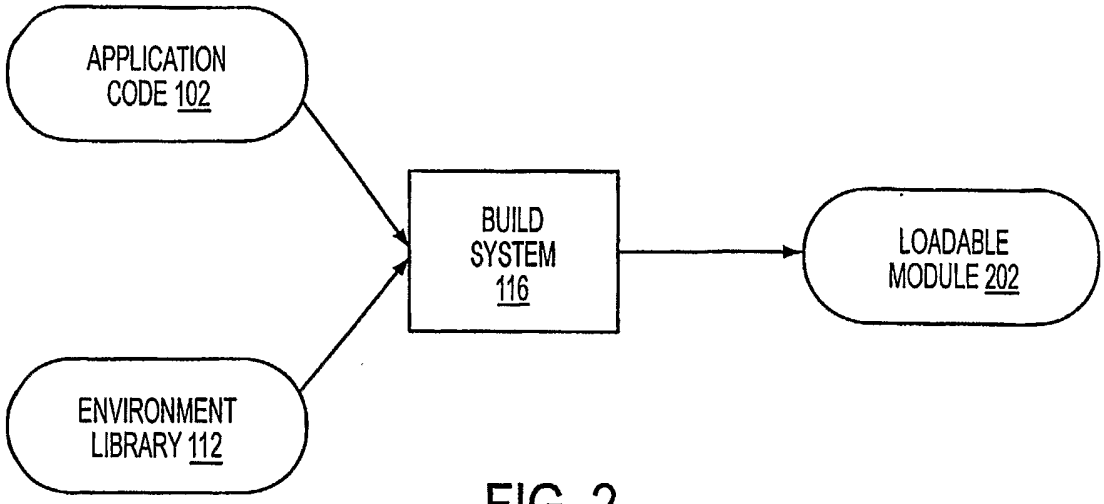

APPLICATION CODE 102
ENVIRONMENT LIBRARY 112
BUILD SYSTEM 116
LOADABLE MODULE 202

FIG. 2

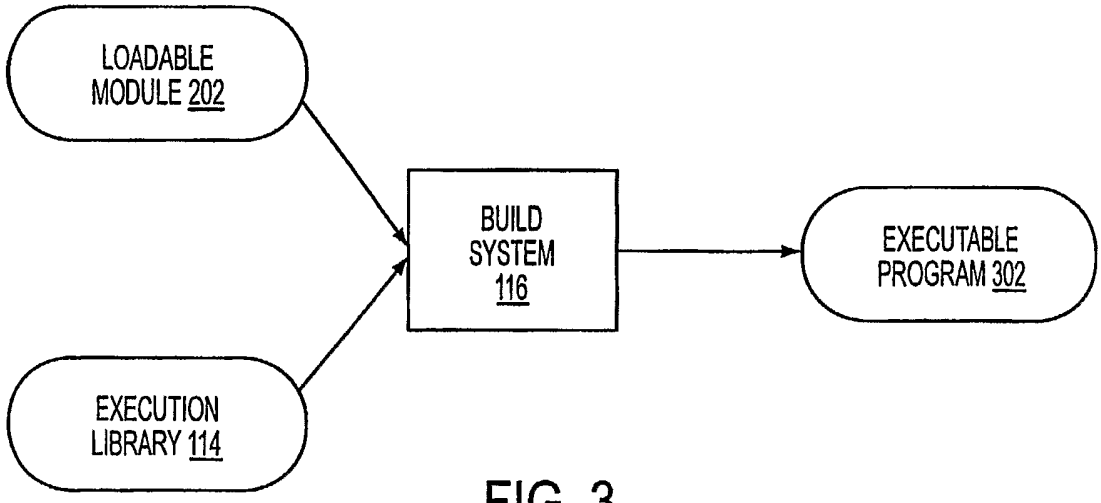

LOADABLE MODULE 202
EXECUTION LIBRARY 114
BUILD SYSTEM 116
EXECUTABLE PROGRAM 302

FIG. 3

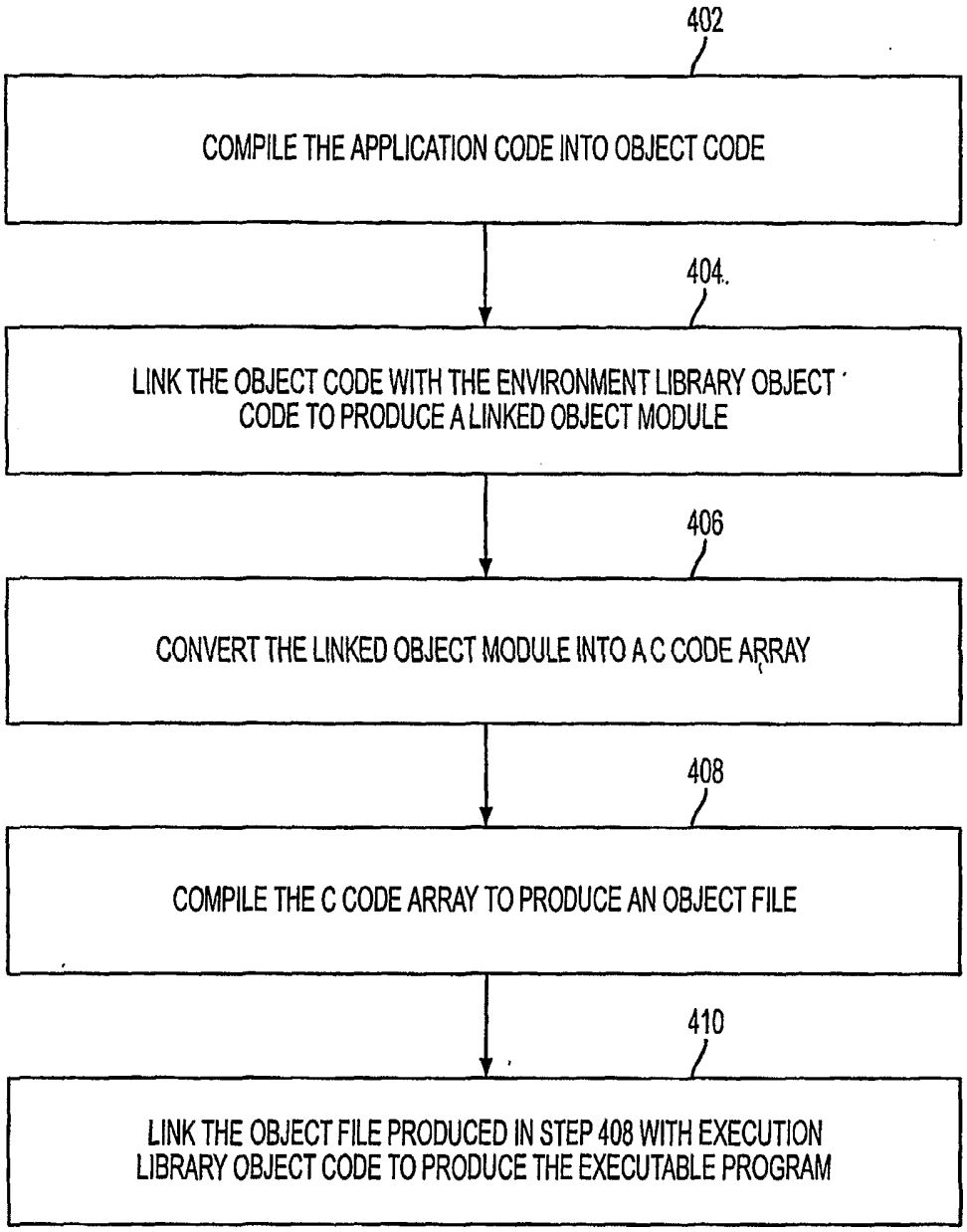

402
COMPILE THE APPLICATION CODE INTO OBJECT CODE
404
LINK THE OBJECT CODE WITH THE ENVIRONMENT LIBRARY OBJECT CODE TO PRODUCE A LINKED OBJECT MODULE
406
CONVERT THE LINKED OBJECT MODULE INTO A C CODE ARRAY
408
COMPILE THE C CODE ARRAY TO PRODUCE AN OBJECT FILE
410
LINK THE OBJECT FILE PRODUCED IN STEP 408 WITH EXECUTION LIBRARY OBJECT CODE TO PRODUCE THE EXECUTABLE PROGRAM

FIG. 4

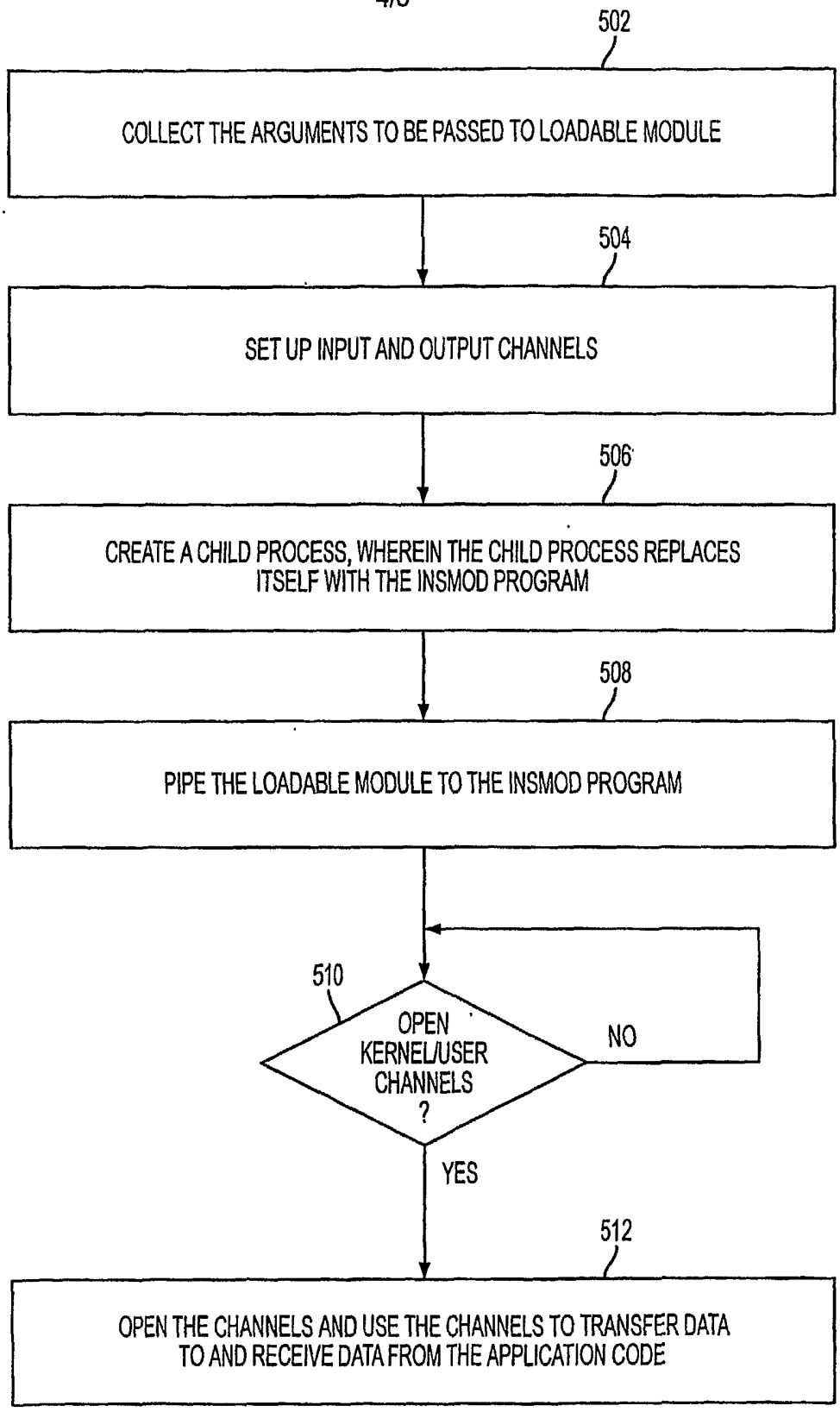

502
COLLECT THE ARGUMENTS TO BE PASSED TO LOADABLE MODULE
504
SET UP INPUT AND OUTPUT CHANNELS
506
CREATE A CHILD PROCESS, WHEREIN THE CHILD PROCESS REPLACES ITSELF WITH THE INSMOD PROGRAM
508
PIPE THE LOADABLE MODULE TO THE INSMOD PROGRAM
510
OPEN KERNEL/USER CHANNELS ?
NO
YES
512
OPEN THE CHANNELS AND USE THE CHANNELS TO TRANSFER DATA TO AND RECEIVE DATA FROM THE APPLICATION CODE

FIG. 5

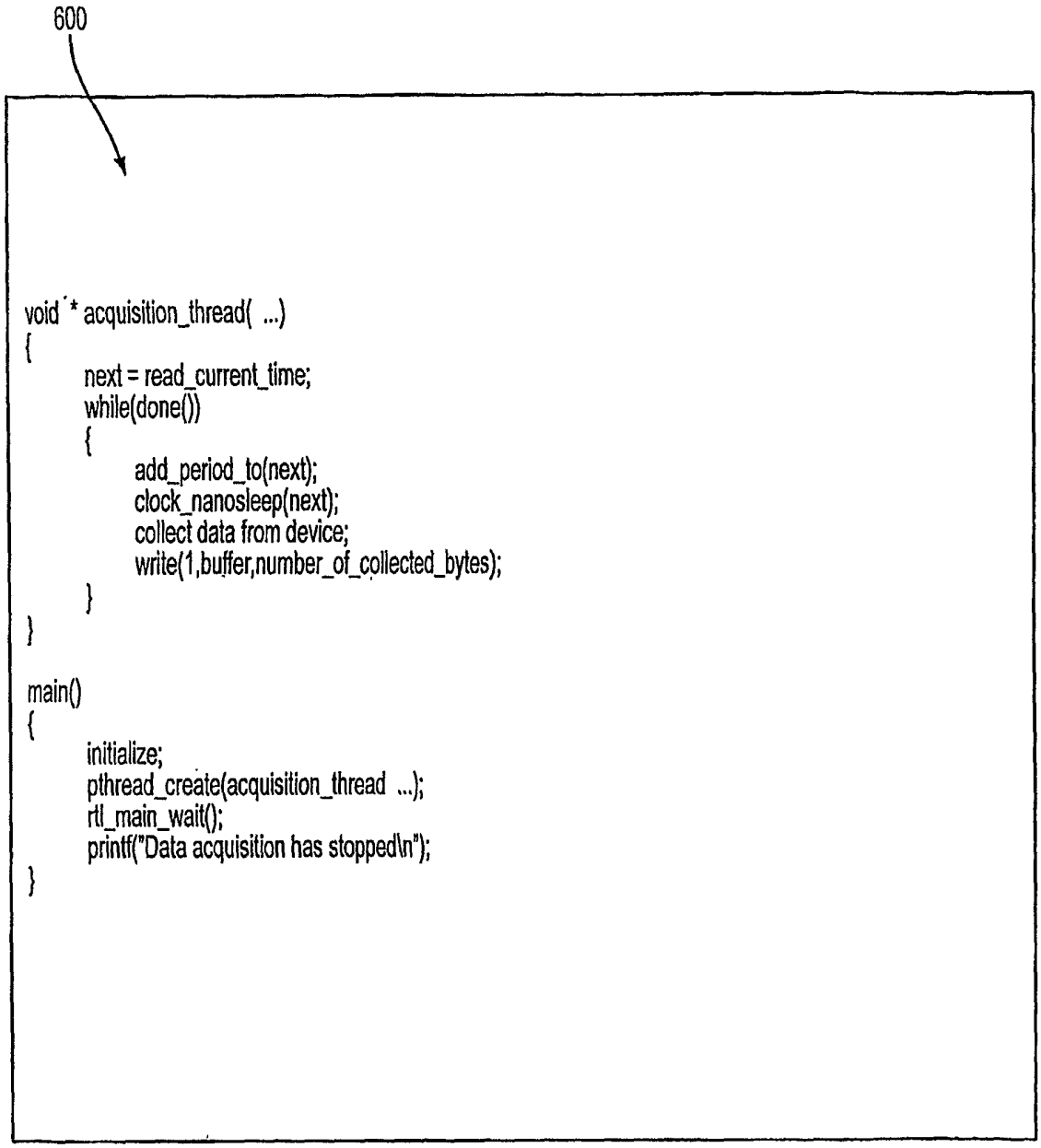

600
void * acquisition_thread(...)
{
next = read_current_time;
while(done())
{
add_period_to(next);
clock_nanosleep(next);
collect data from device;
write(1,buffer,number_of_collected_bytes);
}
}
main()
{
initialize;
pthread_create(acquisition_thread ...);
rtl_main_wait();
printf("Data acquisition has stopped\n");
}

FIG. 6

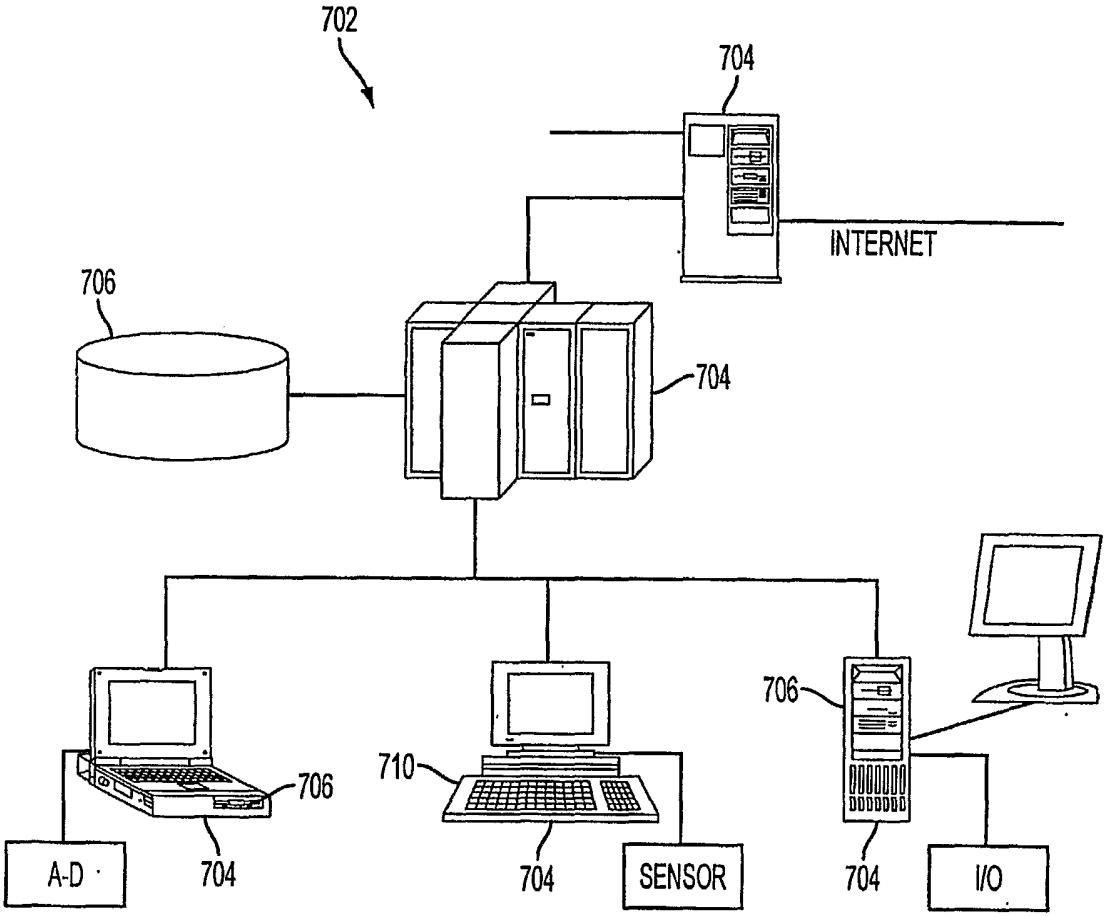


FIG. 7

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US04/31371

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) : G06F 9/44
US CL : 717/106-113,140,162-166
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)
U.S. : 717/106-113,140,162-166

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 practicable, search terms used)
Please See Continuation Sheet

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,136,709 A (SHIRAKABE et al.) 04 August 1992 (04.08.1992), entire reference.	1-66
X	BOS et al, Safe Kernel Programming in the OKE, IEEE, pages 141-152, entire reference, especially Fig. 1, Fig.4, and Fig. 6.	1, 16, 19, 32, 45, 50
X	US 2003/0101290 A1 (LIN et al) 29 May 2003 (29.05.2003), entire reference, especially Figure 1 and Figure 2.	1, 16, 19, 32, 45, 50
A	DORAN, Interfacing Low-Level C Device Drivers with Ada 95, ACM, 1999, pages 133-143, entire reference.	1-66
A	US 6,292,843 B1 (ROMANO) 18 September 2001 (18.09.2001), entire reference.	1-66
A	US 6,463,583 B1 (HAMMOND) 08 October 2002 (08.10.2002), entire reference.	1-66

☐ Further documents are listed in the continuation of Box C. ☐ See patent family 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 application or patent 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
08 December 2004 (08.12.2004)

Date of mailing of the international search report
04 JAN 2005

Name and mailing address of the ISA/US
Mail Stop PCT, Attn: ISA/US
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 22313-1450
Facsimile No. (703) 305-3230

Authorized officer
Ted T. Vo
Telephone No. (571) 272-3706

Form PCT/ISA/210 (second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US04/31371

Continuation of B. FIELDS SEARCHED Item 3:
WEST, ACM, IEEE, CITESEER, GOOGLE
search terms: link, linking, load, loading, operating system kernel, programming environment