UNIT-TESTED OPERATING SYSTEM FOR USE IN KERNEL SPACE AND USER SPACE

Field of the Invention

[0001] The present invention relates generally to the operating systems, and more particularly, to a unit-tested operating system that can be used in the kernel space and the user space for various computer architecture.

Background of the Invention

[0002] An operating system (OS) is a fundamental software component that serves as an intermediary between a computer's hardware and the various software applications that run on it. The OS is a crucial system software that manages and coordinates the hardware resources of a computer, providing an environment in which users and other software can execute tasks efficiently and securely.

[0003] User space and kernel space are two distinct memory spaces in a computer's OS, each serving different purposes and having different levels of privilege and access to system resources. The user space is the memory area where user-level applications and processes run. These applications have limited access to the computer's hardware and must go through the kernel to perform privileged operations. On the other hand, the kernel space is the memory area where the OS's kernel, which is the core part of the operating system, resides. The kernel has full and direct access to the hardware and can execute privileged instructions.

[0004] The operating system (OS) works in the user space by providing a range of services, libraries, and interfaces that allow user-level applications to interact with the underlying hardware and perform various tasks such as executing applications, managing processes, managing memory, accessing file systems, and more. The user space refers to the memory area where normal user programs run. On the other hand, the kernel space of an OS, which is the core part of the OS, is responsible for managing hardware resources, providing services to user-level

applications, ensuring system stability and security, and responding to system events. The kernel space runs in privileged mode and has unrestricted access to the hardware. It handles low-level tasks like memory management, process scheduling, I/O operations, and interfacing between software and hardware. While applications do not have direct access to hardware and must ask the kernel to perform such tasks on their behalf. In other words, the kernel space is the address space reserved for running the kernel, kernel extensions, and depending on the operating system, device drivers. Kernel space operates at the highest privilege level and plays a crucial role in the overall functionality of the operating system. In contrast, the user space is the address space in which user processes (i.e., applications) run. It has restricted privileges and interacts with the kernel space for accessing system resources and hardware. Thus, the kernel space performs critical system tasks and controls access to hardware while the user space is where applications execute and interact with the kernel space services.

[0005] Typically, the kernel space lacks the capability to directly interface with devices and provide the same high-level functionality that the user space side provides. The kernel operates at a low level, managing the hardware and providing basic services like memory management and process scheduling. It does not contain the drivers, protocols, and programming interfaces needed for sophisticated input/output operations. For example, the kernel alone cannot provide a graphical user interface, networking capabilities, or complex file system access. Those facilities are implemented in code running in user space. The user space relies on the kernel to provide lower-level access but contains libraries, frameworks, and applications that actually allow the user to interact with the system. In other words, while the kernel manages and virtualizes the hardware, the user space components build on those services to enable an advanced user experience and application functionality. The separation of concerns between kernel and user space enables both stability and complexity.

[0006] Accordingly, there is an established need for an operating system to be unit-tested and possess the capability of working the same for the user space and the kernel space and to provide a unified library/interface for both the user space side and the kernel space side.

Summary of the Invention

[0007] The present invention is directed to an operating system, i.e., computer readable program instructions, that can work both in the user space and the kernel space. Further, the disclosed computer readable program instructions can be unit-tested. In a first aspect, the disclosed invention is directed to a system including a processor and a non-transitory computer-readable medium with the operating system stored on it that, when executed by the processor, cause the processor to perform operations to modify context of an application to switch between processes in a user space side. The modification can be performed based on a POSIX signal stack.

[0008] In a first implementation of the invention, a computing device comprises a processor; and a non-transitory computer-readable medium having stored thereon instructions that, when executed by the processor, cause the computing device to perform operations including: modifying context of an application to switch between processes in a used space side and enabling interfacing with one or more devices in a kernel space side to provide a kernel space side functionality. The modifying is performed based on a POSIX signal stack and the kernel space side functionality is same as a user space side functionality.

[0009] In another aspect, the instructions are written using C++20 standard to provide both a bare metal kernel solution and a POSIX-compliant user space component that shares the same underlying software components.

[0010] In another aspect, the instructions comprise a core section built on top of an architectural section to provide language features to support C++20 features, affinity, alignment, atomic accesses, bitmaps, strings, clocking, endian support, memory blocks, memory management, priority, locking, and other various components.

[0011] In another aspect, the architectural section comprises a privilege level interface, a register file interface, a stack interface.

[0012] In an aspect, the architectural section further comprises a register file, a stack, abort types, barriers, exclusive memory access, interrupt request types, and execution base class.

[0013] In another aspect, the architectural section is configured to support other architectures through using templates specialization and context expression evaluation.

[0014] In another aspect, the architectural section comprises inline assembly language to remain compatible across different application binary interfaces (ABIs) to support architectures without reliance on specific calling conventions or register assignments.

[0015] In another aspect, the core section is built upon a flat hierarchy and is configured to prevent virtual table modification.

[0016] In another aspect, the architectural section and the core section are centered on memory blocks, pairing size and pointer values.

[0017] In another aspect, the instructions utilize pushing pointers and canaries onto stacks, and bypass a need for static global variables, and knowledge of an ABI by an abstract interface that reads register context information to verify the pointers, canaries, or internal class verification in order to transfer execution into instances of classes.

[0018] In another aspect, the instructions further comprise a process section configured to provide a simplistic task and task signaling mechanism built on top of the architectural section.

[0019] In another aspect, the instructions further comprises a data structure section configured to provide key data structures comprising simplified class templates, C++ language features, fundamental data structures, and device tree implementations.

[0020] In another aspect, the data structure section further comprises clocks, CPUs, scheduling, frequency controllers, interrupt controllers, SMP controllers, and supporting frameworks.

[0021] In another aspect, the instructions are configured to use a linker script to create a final image to be used with bare-metal hardware or in a pre-existing virtual environment.

[0022] In another aspect, the instructions are configured to add an augmentation linker-script to a user space application.

[0023] In another aspect, the instructions are configured to support an ARMv7 architecture.

[0024] In another aspect, the instructions further comprises a test system to unit test individual features and components of the instructions.

[0025] In another aspect, the test system is further configured to provide a framework for testing each individual component, and provide a mechanism to spin up a system to unit test particular sequences of operations, and test active running components.

[0026] In a second implementation, a computer-implemented method comprises modifying context of an application to switch between processes in a used space side and enabling interfacing with one or more devices in a kernel space side to provide a kernel space side functionality. The modifying is performed based on a POSIX signal stack and the kernel space side functionality is same as a user space side functionality.

[0027] In a third implementation, a non-transitory computer-readable medium having computer-executable instructions thereon that, in response to execution by one or more processors of a computing device, cause the computing device to perform actions comprising: modifying context of an application to switch between processes in a used space side and enabling interfacing with one or more devices in a kernel space side to provide a kernel space side functionality. The modifying is performed based on a POSIX signal stack and the kernel space side functionality is same as a user space side functionality.

[0028] These and other objects, features, and advantages of the present invention will become more readily apparent from the attached drawings and the detailed description of the preferred embodiments, which follow.

Brief Description of the Drawings

[0029] The preferred embodiments of the invention will hereinafter be described in conjunction with the appended drawings provided to illustrate and not to limit the invention, where like designations denote like elements, and in which:

[0030] FIG. 1 presents an exemplary diagram of an operating system in communication with one or more software and hardware devices, in accordance with an embodiment of the present invention; and

[0031] FIG. 2 presents an exemplary flowchart of an operating system and the user space and the kernel space, in accordance with an embodiment of the present invention;

[0032] FIG. 3 illustrates a unit-tested operating system with a shared framework, in accordance with an embodiment of the present invention;

[0033] FIG. 4 illustrates the core section and the architecture section of the operating system, in accordance with an embodiment of the present invention;

[0034] FIG. 5 illustrates data structure core and driver core of the operating system in accordance with an embodiment of the present invention;

[0035] FIG. 6 illustrates a unit-tested operating system with a shared framework, in accordance with an embodiment of the present invention;

[0036] Like reference numerals refer to like parts throughout the several views of the drawings.

Detailed Description

[0037] The following detailed description is merely exemplary in nature and is not intended to limit the described embodiments or the application and uses of the described embodiments. As used herein, the word “exemplary” or “illustrative” means “serving as an example, instance, or illustration.” Any implementation described herein as “exemplary” or “illustrative” is not necessarily to be construed as preferred or advantageous over other implementations. All of the implementations described below are exemplary implementations provided to enable persons skilled in the art to make or use the embodiments of the disclosure and are not intended to limit the scope of the disclosure, which is defined by the claims. For purposes of description herein, the terms “upper”, “lower”, “left”, “rear”, “right”, “front”, “vertical”, “horizontal”, and

derivatives thereof shall relate to the invention as oriented in FIG. 1. Furthermore, there is no intention to be bound by any expressed or implied theory presented in the preceding technical field, background, brief summary or the following detailed description. It is also to be understood that the specific devices and processes illustrated in the attached drawings, and described in the following specification, are simply exemplary embodiments of the inventive concepts defined in the appended claims. Hence, specific dimensions and other physical characteristics relating to the embodiments disclosed herein are not to be considered as limiting, unless the claims expressly state otherwise. It should be noted that, the term “comprising” can also encompass the terms “consisting essentially of” and “consisting of”.

[0038] Shown throughout the figures, the present invention is directed toward an operating system that can work both in the user space and the kernel space. Further, the operating system is unit-tested. That is, individual units of source code of the operating system, i.e., sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use in both the kernel space and the user space.

[0039] Generally, there is a line that distinguishes software being on the user space (an application, library, etc.), the kernel space (a kernel itself, drivers, or other operating system components), or the bare metal applications such as boot loaders, firmware, etc. The present invention discloses a unit tested, standalone software that does not require external components and can be added to existing software, which can work in both the user space and the kernel space. To that end, a common framework can be shared between the user space and the kernel space. Unit tests can be built around this framework and the underlying architectural components that allow for the different modes of operation. Due to the shared framework nature, a proper run on the user space can enable correct operation of the kernel space side and vice versa. The disclosed operating system can be self-contained and can test itself, with no additional system being required.

[0040] Referring interchangeably to FIGS. 1 and 2, the operating system, OS, 100 can be in communication with one or more software, e.g., an application 102, or one or more hardware devices, e.g., a disk drive 112, a monitor 104, a keyboard 106, a mouse 110, and a printer 108.

As shown in FIG. 2, the OS 200, which can be similar to the OS 100 shown in FIG. 1, can work at the user space 202 and the kernel space 204. Generally, the operating system (OS) 200 serves as an intermediary between a computer's hardware 206 and the various software applications in the user space 202, such as application 1 208, application 2 210, …, application n 212, that run on the OS 200. The OS 200 can manage and coordinate the hardware resources of the computer in the kernel space 204, e.g., device drivers 218, and the kernel 216, to provide an environment in which users and other software can execute tasks efficiently and securely. In some embodiments, the OS 200 manages hardware resources such as the central processing unit (CPU), memory (e.g., RAM), storage devices (e.g., hard drives and SSDs), and input/output (I/O) devices (e.g., keyboards, mouse, displays). The OS allocates and deallocates these resources to different processes and applications as needed. The OS 100 can be communicatively coupled to a processor 114 and a non-transitory computer-readable medium 116.

The Operating System Overview

[0041] In some embodiments, the OS 100 creates, schedules, and terminates processes or tasks. A process can be a program in execution, and the OS 100 ensures that multiple processes run concurrently, sharing the CPU's time effectively. The OS 100 also provides mechanisms for inter-process communication (IPC). The OS 100 further manages system memory, including allocating memory to processes, tracking memory usage, and handling memory protection to prevent processes from accessing each other's memory space.

[0042] In several embodiments, the OS 100 provides a file system that allows users and applications to create, read, write, and organize files and directories on storage devices. The OS 100 manages file permissions and ensures data integrity. The OS 100 handles communication between software and hardware devices, including device driver management. The OS 100 provides a standardized interface (e.g., an API) for interacting with devices, allowing software to be hardware-independent.

[0043] In an embodiment, the OS 100 offers a user interface 214 through which users interact with the computer. This can be a graphical user interface (GUI) with windows, icons, and menus or a command-line interface (CLI) where users input text commands. The OS 100 enforces security policies and access controls to protect system resources and user data. The OS 100 manages user accounts, authentication, and authorization to ensure that only authorized users can access specific resources. In some embodiments, the OS 100 detects and reports errors and issues that occur during system operation. The OS 100 logs error messages and provides mechanisms for diagnosing and resolving problems.

[0044] The OS 100 can include networking capabilities, allowing computers to connect to networks, share resources, and communicate with other devices over the internet or local networks. Further, in some embodiments, the users can configure various aspects of the OS 100, including display settings, network configurations, and user preferences, to customize their computing environment. The OS 100 can receive updates and patches to fix bugs, enhance security, and add new features. These updates help keep the system stable and secure.

[0045] The OS 100 can include components, even if such components are not being used, such that tests for the ARM architecture components can be run in the user space to verify correct operation for the kernel space side. As such, the OS 100 is a system/library that can be used on both the user space and the kernel space applications for various computer architectures. In some embodiments, the OS 100 is written using C++20 standard. In several embodiments, the core section of the OS 100 can be built on top of the architectural base and provides language features to support C++20 features, affinity, alignment, atomic accesses, bitmaps, strings, clocking, endian support, memory blocks, memory management, priority, locking, and other various components.

[0046] Reference is now made to FIG. 3, in which an architectural section of the operating system is illustrated, in accordance with some embodiments. The architectural section 300 contains the key low-level interfaces, such as privilege level interface 312, register file interface 314, stack interface 316, and components needed for hardware abstraction, including critical structures such as the register file, stack, and exceptions, including abort types 302, barriers 304, exceptions 306, exclusive memory access 308, interrupt request types 310, and execution base

class 318. A modular design of the OS aided by C++ techniques allows it to encapsulate architecture-specific implementation details. The architectural section 300 is designed to support other architectures through extensive use of templates specialization 320 and context expression evaluation 322. This allows compile-time checking for compatibility across architectures. In some embodiments, in addition to templates and constant expression, the architectural section 300 uses carefully written inline assembly language 324 to remain compatible across different ABIs (application binary interfaces). This allows the OS to support various architectures without reliance on specific calling conventions or register assignments.

[0047] While in an embodiment, only the GNU GCC (v12 for C++20 support) is used, in some embodiments, the compiler interface can be abstracted away in the architecture core and be used sparsely to eventually allow for various compilers to be used eventually allowing for a wider variety of hardware architectures that may only have a vendor supplied compiler.

[0048] Further, the architectural and core design can be founded on memory blocks, always pairing a size value with a pointer. Singular pointer usage can be avoided and their usage can be limited to situation when necessary. The blocking strategy can be designed to detect and prevent common software vulnerabilities augmenting classical data structures and containers with a common attack surface. The blocking strategy can be used along with other common vulnerability detection such as canaries on stacks. In some embodiments, memory management blocks are cleaned upon release to prevent data leaks to augment classical representations of data structures such as strings, so that a null terminator is no longer used or necessary.

[0049] Additionally, the data structure core of the OS uses simplified class templates from the core that allow for new features of the C++20 language to be used and provide fundamental data structure types which are common to computer science applications along with an implementation of the device tree standard commonly used as a representation of a system for an operating system.

[0050] In some embodiments, the core section of the OS can be built upon a flat hierarchy to prevent possible virtual table modification, and the use of virtual tables is limited to exceptions where it is necessary to do generally in the driver core, such as extending a clock or other generic

interface for specialized hardware. In various embodiments, a single global static initial memory pool is required which can bypass typical C++ language linkages. As a result, the OS 100 can be compiled as a standalone component as if the OS 100 were to run in kernel space or bare metal hardware. As such, in various embodiments, only user-mode specific code is compiled.

[0051] FIG. 4 illustrates the core section and the architecture section of the operating system, in accordance with some embodiments. In some embodiments, the core section 402 provides higher-level abstractions and language features such as language features 404, affinity 406, alignment 408, atomic access 410, bitmap 412, strings 414, clocking 416, ending support 418, memory block 420, memory management 422, priority 424, and locking 426. The architecture and core design is centered on memory blocks, pairing size and pointer values to enable safer memory usage. As such, the core section 402 above the architectural section 302 (which described above) can provide richer language abstractions and features while relying on the lower level interfaces for hardware access.

[0052] In some embodiments, the OS can utilize pushing pointers and canaries onto stacks, and bypass the need for static global variables, knowledge of an ABI by an abstract interface that reads register context information to verify the pointers, canaries, or internal class verification in order to transfer execution into instances of classes. Further, the process section of the OS can provide a simplistic task and task signaling mechanism that is built on top of the architectural base mechanisms for execution management. In an embodiment, mechanisms are added to extend such task and task signaling mechanism to allow for virtual memory, etc.

[0053] FIG. 5 illustrates data structure core and driver core of the operating system in accordance with some embodiments. The OS can provide key data structures and a device tree representation atop the base cores. The driver core can give frameworks for common OS driver tasks that can be extended as needed. In some embodiments, the data structure core 502 builds on the base core section 402 (as described earlier) by providing key data structures such as simplified class templates 504, C++ language features 506, fundamental data structures 508, and device tree implementations 510. The data structure core 502 can include a device tree implementation for representing system hardware. The driver core 512 can include clocks 514, CPUs 516, scheduling 518, frequency controllers 520, interrupt controllers 522, SMP controllers

524, and supporting frameworks 526. Such components can provide generic driver frameworks that can be specialized for particular hardware.

User Space Component of the OS

[0054] The user space 220 is the memory area where user-level applications and processes run. The OS 100 in the user space is configured to grant the user-level applications limited access to the computer's hardware and make them go through the kernel to perform privileged operations. Thus, user-level applications run in a less privileged mode, and are restricted from performing certain operations directly on hardware, such as accessing memory locations or configuring hardware devices. In some embodiments, the OS in the user space can grant the user-level applications access to system resources, such as files, devices, and network connections, through system calls. These system calls are functions provided by the kernel that act as intermediaries between the user space and the kernel space.

[0055] In some embodiments, the OS in the user space can run the user-level applications with lower privileges, which may result in some performance overhead when accessing system resources through system calls. The OS in the user space can execute the processes running in the user space as isolated processes from each other. As a non-limiting example, if one process crashes or misbehaves, it does not affect other processes. The OS in the user space can grant the user-level processes limited control over system-wide configurations. In some embodiments, the OS in the user space can allow the user-level applications generate and handle errors within their own context. The user-level applications may report errors to the user or log them for debugging purposes. The OS in the user space can run the user-level applications in their own execution contexts, which include memory spaces, CPU registers, and program counters. Context switches occur when switching between user-level processes.

[0056] In various embodiments, the OS can support threading/process execution in the user space 222 by utilizing the POSIX signal stack to modify the context of the application to switch between processes using a scheduler. The OS can support running under user space on the x86-64 architecture POSIX components. In several embodiments, a signal stack, also known as an

alternate signal stack, which is a separate stack used to handle signals asynchronously, is used to notify a process about specific events or conditions, such as the termination of a child process, a user interrupt (e.g., Ctrl+C), or a segmentation fault. When a signal is delivered to a process, it can interrupt the normal flow of execution and execute a signal handler.

[0057] In some embodiments, when a signal is delivered, the current stack's process might not be in a consistent state. By using a separate signal stack, the OS ensures that the signal handler has a clean and predictable stack to execute on, avoiding potential issues related to stack corruption. Even if the process's main stack is full or corrupted, the signal stack provides a dedicated memory space for signal handling. This ensures that signals can be delivered reliably. In an embodiment, the signal stack is a separate memory region allocated by the OS. It is typically smaller in size compared to the main process stack. When a signal is delivered, the OS automatically switches the process's stack to the signal stack before executing the signal handler. After the signal handler completes, the stack is switched back to the main stack.

[0058] In some embodiments, the OS, automatically or by receiving a command from the user, can configure the signal stack's size and location using the sigaltstack system call and the stack_t structure, which allows the users to tailor the signal stack to their application's needs. Below is a simplified, non-limiting example of how to set up a signal stack in C using the sigaltstack function:

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void signal_handler(int signo) {
printf("Signal received: %d\n", signo);
}

int main() {
```

```
// Define a signal stack
  stack_t ss;
 ss.ss_flags = 0;
 ss.ss_size = 8192; // Stack size (adjust as needed)
  ss.ss_sp = malloc(ss.ss_size); // Allocate memory for the signal stack

  if (sigaltstack(&ss, NULL) == -1) {
   perror("sigaltstack");
   return 1;
  }

  // Set up a signal handler
  struct sigaction sa;
  sa.sa_handler = signal_handler;
  sa.sa_flags = SA_ONSTACK; // Use the alternate signal stack

  if (sigaction(SIGUSR1, &sa, NULL) == -1) {
            perror("sigaction");
            return 1;
            }
           // Send a signal to this process
            kill(getpid(), SIGUSR1);

            // Free the allocated memory (cleanup)
```

```
free(ss.ss_sp);


return 0;
}
```

[0059] In this example, the POSIX signal stack is set up and is used to handle the SIGUSR1 signal. When the signal is delivered, the signal_handler function is executed on the alternate signal stack. This ensures that the signal handler has a clean execution environment.

[0060] In some embodiments, the scheduler utilizes the POSIX signals to trigger context switches between processes. Each process is associated with a signal (e.g., SIGUSR1) that is used to interrupt its execution. The scheduler can maintain a list or queue of processes that need to be scheduled for execution. This list can include information about each process, such as its program counter, stack pointer, and signal to be used for context switching. In some embodiments, the scheduler allocates an alternate signal stack for each process. This stack is used exclusively for handling signals and context switches for that process. The stack size should be sufficient to accommodate the needs of the process. Further, when it's time to switch from one process to another, the scheduler can send the appropriate signal to the currently running process. The process receives the signal and executes a signal handler function that performs one or more of the following steps:

[0061] a. Save Context: The signal handler can save the current execution context, including CPU registers and stack pointer, onto the process's signal stack. This saves the state of the currently running process.

[0062] b. Load Context: The signal handler can load the saved context of the next process from its signal stack. This involves restoring CPU registers and the stack pointer to the state of the next process.

[0063] c. Resume Execution: The signal handler can return from the signal handler function, effectively resuming execution of the next process from where it was previously interrupted. The

scheduler has successfully switched between processes. Below is a simplified, non-limiting pseudocode representation of the scheduler's logic:

```
while True:
 # Get the next process to run from the scheduler's queue
  next_process = scheduler.get_next_process()

 # Send a signal to the currently running process to yield the CPU
  send_signal(current_process, SIG_YIELD)

 # Save the current process's context to its signal stack
 save_context(current_process)

 # Set the currently running process to the next process
          current_process = next_process

         # Load the next process's context from its signal stack
          load_context(next_process)

        # Resume execution of the next process
        resume_execution(next_process)
```

[0064] In this simplified pseudocode, the scheduler alternates between processes by sending signals to trigger context switches. Each process has its dedicated signal stack, allowing for clean and isolated context switching.

[0065] Moreover, in some embodiments, the driver core of the OS can be split into operating system components (with eventual expansion) such as, but not limited to clocks, CPUs, scheduling, frequency controllers, interrupt controllers, SMP controllers, and other supporting frameworks. Such components can act as a framework and without the need to be extended to be specialized unless dealing with particular hardware. Specialized hardware classes derive from these fundamental framework classes to provide.

Kernel Space Component of the OS

[0066] The kernel space and bare-metal components can use a standard linker script to create the final image that can be used with bare-metal hardware or in a pre-existing virtual environment like QEMU. Similarly, an augmentation linker-script is added to a user space application if independent modules are used with the system, but is not necessarily required.

[0067] The kernel communicates directly with the computer's hardware, including the central processing unit (CPU), memory, storage devices, input/output (I/O) devices (e.g., keyboard, mouse, display), and network interfaces. The kernel controls and manages these hardware resources to ensure they are utilized efficiently. The kernel manages processes and threads. It schedules and allocates CPU time to different processes, switches between them, and ensures fair access to the CPU. The kernel also handles process creation, termination, and synchronization.

[0068] In some embodiments, the OS in the kernel space is responsible for managing physical and virtual memory. It allocates and deallocates memory for processes, manages virtual memory mappings, and handles page faults to ensure efficient use of RAM. In some embodiments, the OS in the kernel space provides a file system interface, allowing user-level applications to perform file-related operations. The OS in the kernel space manages file data, metadata, and directory structures, enforcing file permissions and ensuring data consistency. Device drivers are kernel modules that enable communication between user-level applications and hardware devices. In some embodiments, the OS in the kernel space loads and manages these drivers, allowing applications to interact with devices transparently.

[0069] In some embodiments, the user-level applications request kernel services and access hardware resources through system calls. These system calls are standardized interfaces provided by the kernel. When a system call is invoked, the CPU switches from the user-mode to the kernel-mode to execute the requested operation with elevated privileges. In some embodiments, the OS in the kernel space handles hardware and software interrupts, which are signals generated by hardware events (e.g., keyboard input, timer) or exceptional conditions (e.g., division by zero). The OS in the kernel space further processes interrupts to respond to events and ensure system stability.

[0070] In some embodiments, the OS in the kernel space enforces access control policies and security mechanisms to protect system resources and data. The OS in the kernel space manages user authentication, authorization, and privilege levels. In some embodiments, the OS in the kernel space monitors system health and handles critical errors and exceptions, and generates error messages, logs, and crash reports, helping administrators diagnose and resolve system issues. In various embodiments, the kernel parameters and configuration settings can be adjusted to customize system behavior. These configurations may affect hardware drivers, resource allocation, and system behavior. In some embodiments, the OS in the kernel space supports loadable kernel modules, which are dynamically loaded and unloaded to extend kernel functionality without requiring a complete kernel rebuild. This allows for greater flexibility and modularity.

[0071] In some embodiments, the OS in the kernel space support virtualization technologies and containerization (e.g., Docker, Kubernetes), enabling the creation of isolated environments for running applications with varying degrees of resource and security isolation. In some embodiments, the OS 100 in the kernel space may be designed for deterministic and predictable behavior to meet strict timing requirements. In some embodiments, the kernel runs in a highly privileged mode. It has full control over the hardware and can execute privileged instructions that are not available to user-level applications. The kernel has direct access to system resources and hardware devices, and manages and controls these resources on behalf of user-level applications. The kernel enforces strict isolation and protection mechanisms to prevent processes

from interfering with each other or accessing unauthorized resources. The kernel maintains the integrity and stability of the system.

[0072] In some embodiments, the kernel is responsible for handling critical system errors and faults, ensuring system stability even when individual user-level applications encounter errors. The kernel operates in a single, privileged execution context and can handle multiple user-level processes and perform context switches when needed. The kernel operates with the highest level of privilege, allowing it to execute critical tasks efficiently and directly access hardware resources.

[0073] In another aspect, the OS in the kernel space can support the ARM side components/drivers to interface with devices to provide the same functionality that the user space side provides. Below are some non-limiting examples of the ARM side components that are supported by the disclosed OS to interface with devices to provide the same functionality that the user space provides. In some embodiments, the ARM side components/drivers are supported to interface with devices to provide the same functionality that the user space side provides. Typically, in an operating system designed for ARM architecture-based devices, there are various components and drivers specific to ARM-based systems. These components and drivers enable the operating system to interact with and manage the hardware effectively.

[0074] In some embodiments, the OS in the kernel space can support various ARM CPU architectures and instruction sets, such as ARMv7, ARMv8 (including 64-bit ARM, also known as AArch64), and custom ARM architectures used in embedded systems. The OS in the kernel space can manage ARM-based devices with different CPU cores from various manufacturers (e.g., ARM Cortex-A, Cortex-R, or Cortex-M cores) using CPU-specific drivers needed to manage these cores efficiently. As a non-limiting example, the OS can support in the kernel space on bare-metal ARMv7 hardware.

[0075] In embodiments where ARM TrustZone technology is used for security and partitioning on ARM-based systems, the OS in the kernel space can include drivers and components to work with TrustZone to isolate secure and non-secure worlds. Further, for devices with TrustZone support, the OS in the kernel space can include drivers and components for

managing secure and non-secure execution environments, cryptographic operations, and secure boot processes. In some embodiments, ARM NEON, which is a SIMD (Single Instruction, Multiple Data) instruction set extension is used for multimedia and signal processing operations. In such embodiments, the OS in the kernel space includes support for NEON and other SIMD extensions when available.

[0076] In some embodiments, ARM-based devices use custom SoCs that integrate CPU cores, GPUs, memory controllers, and various peripherals. In such embodiments, the OS in the kernel space includes drivers for these SoCs to manage their features and hardware components. In some embodiments, the OS in the kernel space includes power management components and drivers to control CPU clock frequencies, voltage levels, and low-power modes to optimize power consumption. In some embodiments, ARM-based devices use GPUs that are designed to work with ARM architectures. In such embodiments, the OS in the kernel space includes graphics drivers tailored to ARM GPUs, enabling hardware-accelerated graphics rendering.

[0077] In some embodiments, ARM system uses ARM Generic Interrupt Controllers (GICs) for managing interrupts. In such embodiments, the OS in the kernel space includes GIC drivers to handle interrupt routing and prioritization. In some embodiments, ARM-based system has timers and clock management units that are critical for scheduling tasks and maintaining system time. In such embodiments, the OS in the kernel space includes drivers for these components. In some embodiments, an ARM-based CPU includes an FPU. In such embodiments, the OS in the kernel space includes support for floating-point arithmetic and may include FPU-specific drivers. In virtualized environments, where an ARM-based system may require components for ARM virtualization extensions (e.g., ARM Virtualization Extensions or ARMv8 Virtualization), the OS in the kernel space allows multiple virtual machines to run on a single ARM-based system. In some embodiments, an ARM-based system can include various peripherals such as UARTs, SPI controllers, I2C controllers, GPIO controllers, USB controllers, and more. In such embodiments, the OS in the kernel space can include drivers for these peripherals to enable communication with external devices.

Unit-Tested OS

[0078] The OS is a unit-tested operating system that works the same for user space and kernel- pace to provide a unified library/interface for both sides. The unit-testing process can be built and used to introduce a new architecture into the ecosystem of the OS with ease, and can be built for bare-metal hardware or as a standard user space application.

[0079] FIG. 6 illustrates a unit-tested operating system with a shared framework, in accordance with some embodiments. The OS 600 can include three different software domains, including the user space 602, the kernel space 504, and the bare metal 606. While the user space 602 and the kernel space 604 can work separately, the user space 602 and the kernel space 604 can interact with each other as shown by the arrow. The OS 600 enables the kernel space 604, such as drivers, to run in the user space 602 by using a shared framework 608 and extensive unit testing 610. This allows the users pace 602 and the kernel space 604 to share code and be tested against each other, which can improve reliability of the OS 600.

[0080] In some embodiments, a test system is spun up within the OS itself, with unit tested components to simulate single stepping through a virtual system to unit test particular features and components of the software itself. As a result, the need for an an external system, container, etc. is eliminated. The testing core of the OS can further provide a framework for testing each individual component of the entire system as well as provide a mechanism to spin up a system within the system to unit test particular sequences of operations, testing active running components of the system itself, or individual components.

[0081] To that end, the OS has undergone rigorous unit testing to ensure its correctness, reliability, and functionality. Each unit-tested component is part of the OS in the kernel or a library that provides essential functionality to user space applications. Non-limiting examples of such components include memory management, file system operations, and device drivers.

[0082] In some embodiments, the software component is designed with a clear separation of concerns and modularity, allowing it to be used in both kernel space and user space contexts. It has well-defined interfaces and APIs for interaction. In some embodiments, unit testing involves breaking down the component's functionality into small, isolated units or functions and

subjecting them to automated tests. These tests check individual functions for correctness and expected behavior. Unit testing ensures that each part of the component functions as intended and can be a critical part of a robust software development process.

[0083] In various embodiments, unit tests are regularly run to detect regressions, ensuring that changes or updates do not introduce new defects or disrupt existing functionality. Unit testing can involve the use of mocks or fakes to isolate the component being tested from external dependencies or hardware interactions. This ensures that the tests focus on the component's behavior without being affected by external factors. Unit tests can be integrated into a continuous integration pipeline, where they are automatically executed whenever changes are made to the component's codebase. This helps catch issues early in the development cycle.

[0084] Comprehensive documentation accompanies the unit-tested component, including API documentation, usage examples, and explanations of how to extend or modify the component if needed. The unit-tested component can be designed with robust error handling and recovery mechanisms to gracefully handle unexpected situations, such as memory allocation failures or hardware errors. In various embodiments, security best practices are followed in the design and implementation of the component to mitigate vulnerabilities and protect against security threats.

[0085] In some embodiments, the OS 600 can enable unit-testing previously challenging unit testable behavior such as register context save/restores, interrupt and exception routines, along with scheduling. Further, the template specialization used in the architectural core can allow for cross-unit testing of architectural components along with compile time selection of sections and methods to enable/disable. This can further allow for flexibility between various sized bus architectures as well as testing interfaces/mechanisms for foreign architectures. C++20 concept requirements can be used with the templates to limit the scope and usage of routines bypassing oftentimes confusing SFINAE.

[0086] In some embodiments the method or methods described above may be executed or carried out by a computing system including a tangible computer-readable storage medium, also described herein as a storage machine, that holds machine-readable instructions executable by a

logic machine (i.e., a processor or programmable control device) to provide, implement, perform, and/or enact the above described methods, processes and/or tasks. When such methods and processes are implemented, the state of the storage machine may be changed to hold different data. For example, the storage machine may include memory devices such as various hard disk drives, CD, or DVD devices. The logic machine may execute machine-readable instructions via one or more physical information and/or logic processing devices. For example, the logic machine may be configured to execute instructions to perform tasks for a computer program. The logic machine may include one or more processors to execute the machine-readable instructions. The computing system may include a display subsystem to display a graphical user interface (GUI) or any visual element of the methods or processes described above. For example, the display subsystem, storage machine, and logic machine may be integrated such that the above method may be executed while visual elements of the disclosed system and/or method are displayed on a display screen for user consumption. The computing system may include an input subsystem that receives user input. The input subsystem may be configured to connect to and receive input from devices such as a mouse, keyboard or gaming controller. For example, a user input may indicate a request that certain task is to be executed by the computing system, such as requesting the computing system to display any of the above described information, or requesting that the user input updates or modifies existing stored information for processing. A communication subsystem may allow the methods described above to be executed or provided over a computer network. For example, the communication subsystem may be configured to enable the computing system to communicate with a plurality of personal computing devices. The communication subsystem may include wired and/or wireless communication devices to facilitate networked communication. The described methods or processes may be executed, provided, or implemented for a user or one or more computing devices via a computer-program product such as via an application programming interface (API).

[0087] Since many modifications, variations, and changes in detail can be made to the described preferred embodiments of the invention, it is intended that all matters in the foregoing description and shown in the accompanying drawings be interpreted as illustrative and not in a limiting sense. Furthermore, it is understood that any of the features presented in the embodiments may be integrated into any of the other embodiments unless explicitly stated

otherwise. The scope of the invention should be determined by the appended claims and their legal equivalents.

What is claimed is:

1. A computing device, comprising:

a processor; and

a non-transitory computer-readable medium having stored thereon instructions that, when executed by the processor, cause the processor to perform operations including:

modifying context of an application to switch between processes in a user space side, wherein the modifying is performed based on a POSIX signal stack; and

enabling interfacing with one or more devices in a kernel space side to provide a kernel space side functionality, wherein the kernel space side functionality is same as a user space side functionality.

UNIT-TESTED OPERATING SYSTEM FOR USE IN KERNEL SPACE AND USER SPACE

Abstract of the Disclosure

A computing device is disclosed which includes a processor, and a non-transitory computer readable medium, and instructions stored in the non-transitory computer readable medium. The executed instructions can cause the processor to modify context of an application to switch between processes in a user space side and enable interfacing with one or more devices in a kernel space side to provide a kernel space side functionality. The kernel space side functionality is same as a user space side functionality, and the modifying is performed based on a POSIX signal stack.