MASTERING LINUX KERNEL DEVELOPMENT PDF, EPUB, EBOOK

Raghu Bharadwaj | 354 pages | 13 Oct 2017 | Packt Publishing Limited | 9781785883057 | English | Birmingham, United Kingdom

Mastering Linux Kernel Development - Mastering Linux Kernel Development [Book]

These are pointers to the parent's task structure. This is a pointer to a list of child task structures. This is a pointer to a list of sibling task structures. This is a pointer to the task structure of the process group leader. All contending processes must be given fair CPU time, and this calls for scheduling based on time slices and process priorities. These attributes contain necessary information that the scheduler uses when deciding on which process gets priority when contending. For normal processes, this field holds a dynamic priority derived from the nice value. Every task belongs to a scheduling entity group of tasks , as scheduling is done at a per-entity level. We will discuss more on these attributes in the next chapter on scheduling.

This field contains information about the scheduling policy of the process, which helps in determining its priority. This field specifies the CPU mask for the process, that is, on which CPU s the process is eligible to be scheduled in a multi-processor system. This field specifies the priority to be applied by real-time scheduling policies. For non-real-time processes, this field is unused. The kernel imposes resource limits to ensure fair allocation of system resources among contending processes.

These limits guarantee that a random process does not monopolize ownership of resources. There are 16 different types of resource limits, and the task structure points to an array of type struct rlimit , in which each offset holds the current and maximum values for a specific resource. During the lifetime of a process, it may access various resource files to get its task done. This results in the process opening, closing, reading, and writing to these files. The system must keep track of these activities; file descriptor elements help the system know which files the process holds. Filesystem information is stored in this field. The file descriptor table contains pointers to all the files that a process opens to perform various operations.

The files field contains a pointer, which points to this file descriptor table. For processes to handle signals, the task structure has various elements that determine how the signals must be handled. These elements identify signals that are currently masked or blocked by the process. This is of type struct sigpending , which identifies signals which are generated but not yet delivered. This field contains a pointer to an alternate stack, which facilitates signal handling. This filed shows the size of the alternate stack, used for signal handling. With current-generation computing platforms powered by multi-core hardware capable of running simultaneous applications, the possibility of multiple processes concurrently initiating kernel mode switch when requesting for the same process is built in. To be able to handle such situations, kernel services are designed to be re-entrant, allowing multiple processes to step in and engage the required services. This mandated the requesting process to maintain its own private kernel stack to keep track of the kernel function call sequence, store local data of the kernel functions, and so on.

The kernel stack is directly mapped to the physical memory, mandating the arrangement to be physically in a contiguous region. The kernel stack by default is 8kb for x and most other bit systems with an option of 4k kernel stack to be configured during kernel build , and 16kb on an x system. To perform such validations, the kernel services must gain access to the task structure of the current process and look through the relevant fields. Similarly, kernel routines might need to have access to the current task structure for modifying various resource structures such as signal handler tables, looking for pending signals, file descriptor table, and memory descriptor among others. However, in register-constricted architectures, where there are few registers to spare, reserving a register to hold the address of the current task structure is not viable.

On such platforms, the task structure of the current process is directly made available at the top of the kernel stack that it owns. This approach renders a significant advantage with respect to locating the task structure , by just masking the least significant bits of the stack pointer. With the evolution of the kernel, the task structure grew and became too large to be contained in the kernel stack, which is already restricted in physical memory 8Kb. As a result, the task structure was moved out of the kernel stack, barring a few key fields that define the process's CPU state and other low-level processor-specific information. This structure is contained on top of the kernel stack and provides a pointer that refers to the current task structure , which can be used by kernel services. The following code snippet shows the implementation of current for an x platform:. This change enables quick access to current process data over looking up the kernel stack.

As this involves changes to low-level architecture code, it has only been implemented for the x architecture, with other architectures planned to follow. Unlike user mode, the kernel mode stack lives in directly mapped memory. When a process invokes a kernel service, which may internally be deeply nested, chances are that it may overrun into immediate memory range. The worst part of it is the kernel will be oblivious to such occurrences.

Kernel programmers usually engage various debug options to track stack usage and detect overruns, but these methods are not handy to prevent stack breaches on production systems. Conventional protection through the use of guard pages is also ruled out here as it wastes an actual memory page. Kernel programmers tend to follow coding standards--minimizing the use of local data, avoiding recursion, and avoiding deep nesting among others--to cut down the probability of a stack breach. However, implementation of feature-rich and deeply layered kernel subsystems may pose various design challenges and complications, especially with the storage subsystem where filesystems, storage drivers, and networking code can be stacked up in several layers, resulting in deeply nested function calls. The Linux kernel community has been pondering over preventing such breaches for quite long, and toward that end, the decision was made to expand the kernel stack to 16kb x, since kernel 3.

Expansion of the kernel stack might prevent some breaches, but at the cost of engaging much of the directly mapped kernel memory for the per-process kernel stack. However, for reliable functioning of the system, it is expected of the kernel to elegantly handle stack breaches when they show up on production systems. With the 4. Since virtual addresses are currently in use to map even a directly mapped page, principally the kernel stack does not actually require physically contiguous pages. The kernel reserves a separate range of addresses for virtually mapped memory, and addresses from this range are allocated when a call to vmalloc is made.

This range of memory is referred as the vmalloc range. Primarily this range is used when programs require huge chunks of memory which are virtually contiguous but physically scattered. Using this, the kernel stack can now be allotted as individual pages, mapped to the vmalloc range. Virtual mapping also enables protection from overruns as a no-access guard page can be allocated with a page table entry without wasting an actual page. Guard pages would prompt the kernel to pop an oops message on memory overrun and initiate a kill against overrunning process. Virtually mapped kernel stacks with guard pages are currently available only for the x architecture support for other architectures seemingly to follow. During kernel boot, a kernel thread called init is spawned, which in turn is configured to initialize the first user-mode process with the same name. The init pid 1 process is then configured to carry out various initialization operations specified through configuration files, creating multiple processes.

Every child process further created which may in turn create its own child process es are all descendants of the init process. Processes thus created end up in a tree-like structure or a single hierarchy model. The shell , which is one such process, becomes the interface for users to create user processes, when programs are called for execution. Fork, vfork, exec, clone, wait and exit are the core kernel interfaces for the creation and control of new process. These operations are invoked through corresponding user-mode APIs. Aptly named, it forks a new process from a running process. When fork succeeds, the new process is created referred to as child by duplicating the caller's address space and task structure. On return from fork , both caller parent and new process child resume executing instructions from the same code segment which was duplicated under copy-on-write.

Fork is perhaps the only API that enters kernel mode in the context of caller process, and on success returns to user mode in the context of both caller and child new process. Most resource entries of the parent's task structure such as memory descriptor, file descriptor table, signal descriptors, and scheduling attributes are inherited by the child, except for a few attributes such as memory locks, pending signals, active timers, and file record locks for the full list of exceptions, refer to the fork 2 man page. Failing to call wait , the child may terminate and be pushed into a zombie state. Duplication of parent process to create a child needs cloning of the user mode address space stack , data , code , and heap segments and task structure of the parent for the child; this would result in execution overhead that leads to un-deterministic process-creation time.

To make matters worse, this process of cloning would be rendered useless if neither parent nor child did not initiate any state-change operations on cloned resources. As per COW, when a child is created, it is allocated a unique task structure with all resource entries including page tables referring to the parent's task structure , with read-only access for both parent and child. Resources are truly duplicated when either of the processes initiates a state change operation, hence the name copy-on-write write in COW implies a state change.

COW does bring effectiveness and optimization to the fore, by deferring the need for duplicating process data until write, and in cases where only read happens, it avoids it altogether. This on-demand copying also reduces the number of swap pages needed, cuts down the time spent on swapping, and might help reduce demand paging. At times creating a child process might not be useful, unless it runs a new program altogether: the exec family of calls serves precisely this purpose. The execve is the system call that executes the program binary file, passed as the first argument to it.

The second and third arguments are null-terminated arrays of arguments and environment strings, to be passed to a new program as command-line arguments. This system call can also be invoked through various glibc library wrappers, which are found to be more convenient and flexible:. Command-line user-interface programs such as shell use the exec interface to launch user-requested program binaries. Unlike fork , vfork creates a child process and blocks the parent, which means that the child runs as a single thread and does not allow concurrency; in other words, the parent process is temporarily suspended until the child exits or call exec. The child shares the data of the parent.

The flow of execution in a process is referred to as a thread , which implies that every process will at least have one thread of execution. Multi-threaded means the existence of multiple flows of execution contexts in a process. With modern many-core architectures, multiple flows of execution in a process can be truly concurrent, achieving fair multitasking. Threads are normally enumerated as pure user-level entities within a process that are scheduled for execution; they share parent's virtual address space and system resources.

Each thread maintains its code, stack, and thread local storage. Threads are scheduled and managed by the thread library, which uses a structure referred to as a thread object to hold a unique thread identifier, for scheduling attributes and to save the thread context. Mastering linux kernel development : a kernel developer's reference manual. Mastering Linux Kernel Development. All rights reserved. Please sign in to WorldCat Don't have an account? Remember me on this computer. Cancel Forgot your password? Year Language English. Mastering Linux Kernel development : a kernel developer's reference manual by Raghu Bharadwaj;. Print book. Comprehending Processes Address Space and Threads. Deciphering the Process Scheduler. Signal Management. Memory Management and Allocators. Filesystems and File IO. Interprocess Communication. Virtual Memory Management. Kernel Synchronization and Locking. Interrupts and Deferred Work.

Clock and Time Management. Module Management. Bibliografische Informationen.

Holdings: Mastering Linux Kernel development :

Leverage the power of Linux to develop captivating and powerful embedded Linux projects About This Book …. Skip to main content. Start your free trial. Book description Explore Implementation of core kernel subsystems About This Book Master the design, components, and structures of core kernel subsystems Explore kernel programming interfaces and related algorithms under the hood Completely updated material for the 4. What You Will Learn Comprehend processes and fles - the core abstraction mechanisms of the Linux kernel that promote effective simplification and dynamism Decipher process scheduling and understand effective capacity utilization under general and real-time dispositions Simplify and learn more about process communication techniques through signals and IPC mechanisms Capture the rudiments of memory by grasping the key concepts and principles of physical and virtual memory management Take a sharp and precise look at all the key aspects of interrupt management and the clock subsystem Understand concurrent execution on SMP platforms through kernel synchronization and locking techniques In Detail Mastering Linux Kernel Development looks at the Linux kernel, its internal arrangement and design, and various core subsystems, helping you to gain significant understanding of this open source marvel.

Style and approach Each chapter begins with the basic conceptual know-how for a subsystem and extends into the details of its implementation. Seite Inhalt Preface. Comprehending Processes Address Space and Threads. Deciphering the Process Scheduler. Signal Management. Memory Management and Allocators. Filesystems and File IO. Interprocess Communication. Virtual Memory Management. Kernel Synchronization and Locking. Interrupts and Deferred Work. Clock and Time Management. To augment the need for running background operations, the kernel spawns threads similar to processes. These kernel threads are similar to regular processes, in that they are represented by a task structure and assigned a PID.

Unlike user processes, they do not have any address space mapped, and run exclusively in kernel mode, which makes them non-interactive.

Various kernel subsystems use kthreads to run periodic and asynchronous operations. All kernel threads are descendants of kthreadd pid 2 , which is spawned by the kernel pid 0 during boot. The kthreadd enumerates other kernel threads; it provides interface routines through which other kernel threads can be dynamically spawned at runtime by kernel services. Kernel threads can be viewed from the command line with the ps -ef command--they are shown in [square brackets]:. It then wakes up kthreadd and waits for thread creation to complete:.

This routine creates the thread and signals completion:. The following figure sums up the call sequence for process creation:. During the lifetime of a process, it traverses through many states before it ultimately terminates. Users must have proper mechanisms to be updated with all that happens to a process during its lifetime. Linux provides a set of functions for this purpose. This can be achieved using the wait family of system calls:. These system calls update the calling process with the state change events of a child. The following state change events are notified:. In addition to reporting the status, these APIs allow the parent process to reap a terminated child. A process on termination is put into zombie state until the immediate parent engages the wait call to reap it. Every process must end. Process termination is done either by the process calling exit or when the main function returns.

A process may also be terminated abruptly on receiving a signal or exception that forces it to terminate, such as the KILL command, which sends a signal to kill the process, or when an exception is raised. Upon termination, the process is put into exit state until the immediate parent reaps it. Users logged into a Linux system have a transparent view of various system entities such as global resources, processes, kernel, and users. For instance, a valid user can access PIDs of all running processes on the system irrespective of the user to which they belong. Users can observe the presence of other users on the system, and they can run commands to view the state of global system global resources such as memory, filesystem mounts, and devices. However, such transparency is unwarranted on a few server platforms. For instance, consider cloud service providers offering PaaS platform as a service.

They offer an environment to host and deploy custom client applications. They manage runtime, storage, operating system, middleware, and networking services, leaving customers to manage their applications and data. PaaS services are used by various e-commerce, financial, online gaming, and other related enterprises. For efficient and effective isolation and resource management for clients, PaaS service providers use various tools.

They virtualize the system environment for each client to achieve security, reliability, and robustness. The Linux kernel provides low-level mechanisms in the form of cgroups and namespaces for building various lightweight tools that can virtualize the system environment. Docker is one such framework that builds on cgroups and namespaces. Namespaces fundamentally are mechanisms to abstract, isolate, and limit the visibility that a group of processes has over various system entities such as process trees, network interfaces, user IDs, and filesystem mounts.

Namespaces are categorized into several groups, which we will now see. Traditionally , mount and unmount operations will change the filesystem view as seen by all processes in the system; in other words, there is one global mount namespace seen by all processes. The mount namespace s confine the set of filesystem mount points visible within a process namespace, enabling one process group in a mount namespace to have an exclusive view of the filesystem list compared to another process. These enable isolating the system's host and domain name within a uts namespace. This makes initialization and configuration scripts able to be guided based on the respective namespaces.

This prevents one process from an ipc namespace accessing the resources of another. The PID namespace allows a process to spin off a new tree of processes under it with its own root process PID 1 process. PID namespaces are used in containers lightweight virtualization solution to migrate a container with a process tree, onto a different host system without any changes to PIDs. This type of namespace provides abstraction and virtualization of network protocol services and interfaces. Each network namespace will have its own network device instances that can be configured with individual network addresses.

Isolation is enabled for other network services: routing table, port number, and so on. User namespaces allow a process to use unique user and group IDs within and outside a namespace. This means that a process can use privileged user and group IDs zero within a user namespace and continue with non-zero user and group IDs outside the namespace. Processes inside a cgroup namespace are only able to view paths relative to their namespace root. Cgroups are kernel mechanisms to restrict and measure resource allocations to each process group. Using cgroups, you can allocate resources such as CPU time, network, and memory. Similar to the process model in Linux, where each process is a child to a parent and relatively descends from the init process thus forming a single-tree like structure, cgroups are hierarchical, where child cgroups inherit the attributes of the parent, but what makes is different is that multiple cgroup hierarchies can exist within a single system, with each having distinct resource prerogatives.

Applying cgroups on namespaces results in isolation of processes into containers within a system, where resources are managed distinctly. Each container is a lightweight virtual machine, all of which run as individual entities and are oblivious of other entities within the same system. The following are namespace APIs described in the Linux man page for namespaces :. We understood one of the principal abstractions of Linux called the process , and the whole ecosystem that facilitates this abstraction. The challenge now remains in running the scores of processes by providing fair CPU time. With many-core systems imposing a multitude of processes with diverse policies and priorities, the need for deterministic scheduling is paramount. In our next chapter, we will delve into process scheduling, another critical aspect of process management, and comprehend how the Linux scheduler is designed to handle this diversity.

Raghu Bharadwaj is a leading consultant, contributor, and corporate trainer on the Linux kernel with experience spanning close to two decades. He is an ardent kernel enthusiast and expert, and has been closely following the Linux kernel since the late 90s. He is the founder of TECH VEDA, which specializes in engineering and skilling services on the Linux kernel, through technical support, kernel contributions, and advanced training. His precise understanding and articulation of the kernel has been a hallmark, and his penchant for software designs and OS architectures has garnered him special mention from his clients. Raghu is also an expert in delivering solution-oriented, customized training programs for engineering teams working on the Linux kernel, Linux drivers, and Embedded Linux. A comprehensive guide to programming with network sockets, implementing internet protocols, designing IoT devices, and much more with C.

About this book Mastering Linux Kernel Development looks at the Linux kernel, its internal arrangement and design, and various core subsystems, helping you to gain significant understanding of this open source marvel. Publication date: October Publisher Packt. Pages ISBN Chapter 1. Comprehending Processes, Address Space, and Threads. The illusion called address space. Kernel and user space. Note System calls are the kernel's interfaces to expose its services to application processes; they are also called kernel entry points. Process context. Process descriptors. Process attributes - key elements. While in this wait state, any signals generated for the process are delivered, causing it to wake up before the wait condition is met. This process state is rarely used. Process relations - key elements. Scheduling attributes - key elements. Process limits - key elements. File descriptor table - key elements. Signal descriptor - key elements.

Kernel stack. The issue of stack overflow. Process creation. Copy-on-write COW. Linux support for threads. Kernel threads. Process status and termination. Termination of child Stopped by a signal Resumed by a signal. Takes the exit code returned by the child to the parent.

Mastering Linux Kernel Development [Book]

Mastering Linux Kernel Development looks at the Linux kernel, its internal arrangement and design, and various core subsystems, helping you to gain significant understanding of this open source marvel. You will look at how the Linux kernel, which possesses a kind of collective intelligence thanks to its scores of contributors, remains so elegant owing to its great design. This book also looks at all the key kernel code, core data structures, functions, and macros, giving you a comprehensive foundation of the implementation details of the kernel's core services and mechanisms. You will also look at the Linux kernel as well-designed software, which gives us insights into software design in general that are easily scalable yet fundamentally strong and safe. By the end of this book, you will have considerable understanding of and appreciation for the Linux kernel. Each chapter begins with the basic conceptual know-how for a subsystem and extends into the details of its implementation.

We use appropriate code excerpts of critical routines and data structures for subsystems. Galvin, Greg Gagne. Leverage the power of Linux to develop captivating and powerful embedded Linux projects About This Book …. Skip to main content. Updating results WorldCat is the world's largest library catalog, helping you find library materials online. Don't have an account? You can easily create a free account. Your Web browser is not enabled for JavaScript. Some features of WorldCat will not be available. Create lists, bibliographies and reviews: or. Search WorldCat Find items in libraries near you.

Advanced Search Find a Library. Showing all editions for 'Mastering Linux Kernel Development. Refine Your Search Year. Displaying Editions 1 - 9 out of 9. Your list has reached the maximum number of items.

Formats and Editions of Mastering Linux Kernel Development. []

Explore a preview version of Mastering Linux Kernel Development right now. If you are a kernel programmer with a knowledge of kernel APIs and are looking to build a comprehensive understanding, and eager to explore the implementation, of kernel subsystems, this book is for you. It sets out to unravel the underlying details of kernel APIs and data structures, piercing through the complex kernel layers and gives you the edge you need to take your skills to the next level.

Mastering Linux Kernel Development looks at the Linux kernel, its internal arrangement and design, and various core subsystems, helping you to gain significant understanding of this open source marvel. You will look at how the Linux kernel, which possesses a kind of collective intelligence thanks to its scores of contributors, remains so elegant owing to its great design. This book also looks at all the key kernel code, core data structures, functions, and macros, giving you a comprehensive foundation of the implementation details of the kernel's core services and mechanisms. You will also look at the Linux kernel as well-designed software, which gives us insights into software design in general that are easily scalable yet fundamentally strong and safe. By the end of this book, you will have considerable understanding of and appreciation for the Linux kernel. Each chapter begins with the basic conceptual know-how for a subsystem and extends into the details of its implementation. We use appropriate code excerpts of critical routines and data structures for subsystems.

Refine Your Search Year. Displaying Editions 1 - 9 out of 9. Your list has reached the maximum number of items. Please create a new list with a new name; move some items to a new or existing list; or delete some items. Mastering Linux Kernel development : a kernel developer's reference manual. Mastering linux kernel development : a kernel developer's reference manual. Mastering Linux Kernel Development. All rights reserved. Please sign in to WorldCat Don't have an account? Remember me on this computer. Cancel Forgot your password? Year Language English. Mastering Linux Kernel development : a kernel developer's reference manual by Raghu Bharadwaj;.

Labanotation for Beginners download PDF
The Young Errol : Flynn Before Hollywood pdf, epub, mobi
55 Large Print Word Search Puzzles and Solutions : Word Game Easy Quiz Books for Beginners (Find a W download PDF