# Binder ipc driver linux android

In Linux, there are several methods for achieving IPC (interspeed communication), such as files, sockets, signals, pipes, message queues, semaphores, shared memory, etc. This way, the data can be transferred to a remote call method and the results can be returned to the customer's call stream. It seems that the flow from the client process goes into another (remote) process and starts performing there (known as Thread Migration).hat is Binder? IPC/component system to develop object-oriented os servicesA driver core to facilitate intersusion communication. Light RPC (remote communication procedure). A pool of threads to process queries. Synchronous b/w processesNo another object-oriented core. Instead, an object-oriented operating system environment that works on traditional cores such as Linux! Necessary for Android! Comes from OpenBinder:-Started at Be, Inc. as a key part of the next generation of BeOS (no 2001) acquired by PalmSource, The first implementation used in Palm Cobalt (micro-core-based OS), Palm switched to Linux, so That Binder is ported to Linux, open source (No 2005), Google has hired Diane Hackborn, a key engineer of OpenBinder, to join the Android team Used as is for the original cast in Android, but then completely rewritten (No 2008) OpenBinder is no longer supported - long live Binder! Intersy-spanish communication (IPC) is the basis for the exchange of signals and data between multiple processes. Used to send messages, synchronize, shared memory, and remote procedures calls (RPC). Allows information sharing, computational acceleration, modularity, convenience, privilege separation, data isolation, stability (Each process has its own (sandbox) address space, usually works under a unique ID system)Many IPC settings:-Files (including memory displayed)Signals Sockets (UNIX domain, TCP/IP)Pipes (including named pipes) SphoresShares Binder This term is used ambiguously. Binder refers to the overall Binder architecture, while Binder refers to the specific implementation of the Binder interface. Binder Object is a class copy that implements the Binder interface. Binder can implement several Binders.Binder Protocol Binder Medium software uses a very low-level protocol to communicate with the driver. Интерфейс IBinder A is a well-defined set of methods, properties, and events that Binder can implement. That's the way it is. described in aidL1. Binder Token Is a Numerical Value that uniquely identifies Binder.Android apps and system services run in separate processes for security, stability and memory management reasons, but they need to communicate and share data! Security: Each process is in the sandbox and operates under a clear system identity, Stability: if the process behaves improperly (such as a glitch), it does not affect any other memory management processes: unverified processes are removed for free resources (mostly memory) for new, in fact, one Android application can have its components running in separate processes. IPC is on the aid But we need to avoid the overheads of traditional IPC and avoid denial of service issues. Android libc (aka bionic) does not support system V IPCs, no SysV semaphores, general memory segments, message queues, etc. Binder to help! The built-in reference counting of references to an object plus a death notification mechanism makes it suitable for hostile environments (where a low-memory killer roams), when the binder's service is no longer mentioned by any customers, its owner is automatically notified that he can dispose of it. Many other functions: Thread migration - for example, programming model: Automatic management of methods of pools of threads on remote objects can be caused as if they were local - the flow seems to be move on to another process synchronous and asynchronous (one way) model of call Identity Receivers (via UID/PID) - important for security reasons Unique object-display across the process is the ability to send file handles across the boundaries of the Simple Android Interface Definition Language (AIDL) Interface Interface (AIDL) Built-in support to marshal many common data types Simplified model Call transactions through automatically generated proxies and stubs (Java only) Recursion through processes - i.e. behaves in the same way as recursion semantics when calling local execution methods (no IPC/data marshal) But: No support for RPC (local only) Customer service message on Based on communication - not well suited for streaming is not defined by POSIX or any other standard. Most applications and basic system services depend on binder Most components of the feedback call lifecycle application (e.g. onResume (e.g., onDestory () etc.) are called by ActivityManagerService via the link to turn off the link, and the entire system grinds to a halt (no display, no audio, no input, no sensors,...) in some Unix (e.g. RILD) outlets are used. As for the customer, he just wants to Service: While processes can't directly call operations (or read/write data) to other processes, the kernel can, so they use the Binder driver: Because the service can receive simultaneous requests from multiple customers, it should protect (synchronize access to) its mutated state. Binder Framework is a client-server model. Each client initiates communication and waits for a response from the server. Each customer will have a proxy to communicate with the customer. The side of the server is a pool of workflows. The server should generate a new thread for each new Binder request from the customer. The bridge between the client and the server process is the driver of the link. The Binder driver is a character device that is part of the core space. This module ensures that the customer reaches the appropriate destination across the boundaries of the process. If you want to know more about this or any request, please leave a comment here! Thank you for your support! The introduction of Android Binder is a mechanism of intersesy communication (IPC). It is widely used in all Android devices. The driver of the binder kernel has been present in the Linux kernel upstream for quite some time. Binder has been a controversial patchet (see this lwn article as an example). Its design was deemed incorrect and violated certain principles of kernel design (for example, the task should never touch another task file handle table). Most core developers were not fans of the binder. Recently, the upstream link code has fortunately been significantly redesigned (for example, it no longer touches another table of task file handles, the lock is now very fine-grained, etc.). Since Android is one of the main operating systems (OS) for a huge number of devices there is simply no way around the binder. The Android IPC Link Service Manager mechanism is available from the user space through device nodes located on /dev. The modern Android system will highlight three nodes of the device: /dev/binder/dev/hwbinder /dev/vndbinder, serving different purposes. However, the logic is the same for all three of them. The process can cause the device to open (2) on these nodes to obtain FD, which it can then use to issue requests through ioctl (2)s. Android has a service manager that is used to translate addresses into bus names, and only the address of the service manager is well known. The service manager is registered through ioctl (2) and there can only be one service manager. This means that once the service manager has taken possession of the binder devices, they cannot be (easily) reused by a second service manager. Running Android in containers It matters as soon as a few Android copies are to be launched. Because they will need their own private binders. This is a use case that occurs quite naturally when running Android in system containers. People have been doing this for a long time with The project that intended to make the launch of Android in LXC containers is very easy Anbox. Anbox allows you to run hundreds of Android containers. To properly run Android in a container, you need each container to have a set of private binders. Statically Styling Binder Binder Devices are currently statically distributed during compilation. Before compiling the kernel, CONFIG_ANDROID_BINDER_DEVICES must place a bet set in the kernel (Kconfig) that contains the names of binders to highlight when downloaded. By default, it is set as: CONFIG_ANDROID_BINDER_DEVICES binder, hwbinder, vndbinder To highlight additional binders, the user needs to specify them using this option Kconfig. This is problematic because users need to know how many containers they will run at their maximum and then calculate the number of devices they need so they can specify them in Kconfig. When the maximum number of binder devices required changes after the kernel is compiled, the only way to get additional devices is to recompulate the kernel. Problem 1: Using the wrong basic device number This situation is exacerbated by the fact that binders use the main misc number in the nucleus. Each site in the Linux kernel is identified by a large and insignificant number. The device can request its own main number. If it does this will have an exclusive range of insignificant numbers it does not share with anything else and is free to give away. Or he can use the wrong large number. The main misc number is common among different devices. However, this also means that the number of minor devices that can be handed out is limited to all users misc major. So if a user requests a very large number of binders in Kconfig they can make it impossible for anyone else to highlight the minor numbers. Or just might not be enough to highlight for yourself. Problem 2: Containers and IPC namespaces All of these binders requested by Kconfig through CONFIG_ANDROID_BINDER_DEVICES will be highlighted when downloaded and placed in devtmpfs hosts, usually located in/dev or - depending on the implementation of udev (7) - will be created through mknod (2) - udev (7) when downloaded. This means that all of these devices originally belong to the IPC namespace host. However, containers usually work in their own IPC name space separately from the host. But when the binder devices located in/dev are transferred to containers (e.g., with binding-mounting), the core driver will not know that these devices are now used in another IPC name space because the driver is not aware of the IPC name space. It's not a serious technical it's a serious conceptual one. There has to be a way to have the name binder device on the IPC. Enter binderfs To solve both problems we came up with a solution that I presented at the Linux Plumbers Conference in Vancouver this year. There's a video of that presentation presentation на Youtube: Android binderfs - это крошечная файловая система, которая позволяет пользователям динамически распределять связующие устройства, т.е. позволяет добавлять и удалять связующие устройства во время выполнения. Это означает, что он решает проблему 1. Кроме того, устройства связующего звена, расположенные в новом экземпляре binderfs, не зависит от связующих устройств, расположенных в другом экземпляре binderfs. Все устройства связующего устройства в экземплярах binderfs также не зависит от связующих устройств, выделенных во время загрузки, указанной CONFIG_ANDROID_BINDER_DEVICES. Это означает, что binderfs решает проблему 2. Android binderfs могут быть установлены через: монтировать-t связующего связующего /dev/binderfs в этот момент новый экземпляр binderfs появится на /dev/binderfs. В новом экземпляре binderfs никаких связующих устройств не будет. Там будет только связующее устройство управления, который служит обработчиком запросов для binderfs: root@edfu: ls -al /dev/binderfs/ Всего 0 drwxr-xr-x 2 корень корня 0 Январь 10 15:07 . drwxr-xr-x 20 корень корня 4260 Январь 10 15:07 .. crw------- 1 корень корня 242, 6 января 10 15:07 связующего управления binderfs: Динамически Распределение нового устройства связующего для выделения нового устройства связующего устройства в случае связующего устройства запрос должен быть отправлен через узел устройства управления связующим звеном. Запрос отправляется в виде ioctl(2). Вот пример программы: #define _GNU_SOURCE #include &lt;errno.h&gt;#include #include #include #include #include #include #include #include &lt;fcntl.h&gt;#include &lt;stdio.h&gt; &lt;stdlib.h&gt; &lt;string.h&gt; &lt;sys octl.h=&gt; &lt;sys tat.h=&gt; &lt;sys ypes.h=&gt; &lt;unistd.h&gt; &lt;linux ndroid/binder.h=&gt; &lt;linux ndroid/binderfs.h=&gt;int main (int argc, char s argv) ret, saved_errno; size_t len; struct binderfs_device устройство No 0; если (argc !) 3) выход (EXIT_FAILURE); len s strlen (argv);; если (len &gt; BINDERFS_MAX_NAME) выход (EXIT_FAILURE); memcpy (device.name, argv'2,len); f O_RDONLY d O_CLOEXEC); если &lt; 0)= {= printf(%s= -= failed= to= open= binder-control= device,= strerror(errno));= exit(exit_failure);= }= ret=ioctl(fd, binder_ctl_add,= &amp;device);= saved_errno=errno; close(fd);= errno=saved_errno; if= (ret=&gt; &lt; 0) { printf(%s - Failed to allocate new binder device, strerror(errno)); exit(EXIT_FAILURE); } printf(Allocated new binder device with major %d, minor %d, and name %s, device.major, device.minor, device.name); exit(EXIT_SUCCESS); } What this program simply does is to open the binder-control device node and sending a BINDER_CTL_ADD request to the kernel. Users of binderfs need to tell the kernel which name the new binder device should get. By default a name can only contain up to 256 chars including the terminating zero byte. The struct which is used is: /** * struct - retrieve information about a new binder device * @name: the name to use for the new binderfs binder device * @major: major number allocated for binderfs binder devices * @minor: minor number allocated for the new binderfs binder device * */ struct 0)= {= printf(%s= -= failed= to= allocate= new= binder= device,= strerror(errno));= exit(exit_failure);= }= printf(allocated= new= binder= device= with= major= %d,= minor= %d,= and= = name= %s,= device.major,= device.minor,= device.name);= exit(exit_success);= }= what= this= program= simply= does= is= to= open= the= binder-control= device= node= and= sending= a= binder_ctl_add= request= to= the= kernel.= users= of= binderfs= need= to= tell= the= kernel= which= name= the= new= binder= device= should= get.= by= default= a= name= can= only= contain= up= to= 256= chars= including= the= terminating= zero= byte.= the= struct= which= is= used= is := *= *= struct= binderfs_device= -= retrieve= information= about= a= new= binder= device= *= @name:= the= name= to= use= for= the= new= binderfs= binder= device= *= @major:= major= number= allocated= for= binderfs= binder= devices= *= @minor:= minor= number= allocated= for= the= new= binderfs= binder= device= *= */= struct=&gt;&lt;/ 0) { printf(%s - Failed to allocate new binder device, strerror(errno)); exit(EXIT_FAILURE); } printf(Allocated new binder device with major %d, minor %d, and name %s, device.major, device.minor, device.name); exit(EXIT_SUCCESS); } What this program simply does is to open the binder-control device node and sending a BINDER_CTL_ADD request to the kernel. Users of binderfs need to tell the kernel which name the new binder device should get. By default a name can only contain up to 256 chars including the terminating zero byte. The struct which is used is: /** * struct binderfs_device - retrieve information about a new binder device * @name: the name to use for the new binderfs binder device * @major: major number allocated for binderfs binder devices * @minor: minor number allocated for the new binderfs binder device * */ struct &gt; (fd&lt;/linux&gt; &lt;/linux&gt; &lt;/unistd.h&gt; &lt;/sys&gt; &lt;/sys&gt; &lt;/sys&gt; &lt;/string.h&gt; &lt;/stdlib.h&gt; &lt;/stdio.h&gt; &lt;/fcntl.h&gt; &lt;/errno.h&gt; &lt;/errno.h&gt; - char name (BINDERFS_MAX_NAME 1); __u32 major; __u32 a minor; and is defined in linux/android/binderfs.h. Once the request is made through the ioctl (2) passing structure of the binder_device with the name to the core it will highlight the new device device and return a major and insignificant number of new device in the structure (This is necessary because the binderfs highlighted the main device number dynamically on the shoe.). Upon the return of the ioctl (2) there will be a new binder device located under /dev/binderfs with the selected name: root@edfu: ls-al/dev/binderfs/ just 0 drwxr-xr-x 2 Root Root Root 0 Jan 10 15:19 . drwxr-xr-x 20 Root Root 4260 Jan 10 15:07 .. crw------- 1 root root 242, 0 January 10 15:19 binding control crw------- 1 root root 242, 1 January 10 15:19 my-binder crw------- 1 root root 242, January 2 10 15:19 my-binder1 binderfs: Removal of the binder device does not involve the issuance of another request ioctl (2) through binding control. They can be removed through unlink (2). This means that the rm(1) tool can be used to remove them: root@edfu: rm/dev/binderfs/my-binder1 root@edfu: ls-al/dev/binderfs/ Total 0 drwxr-xr-xr-x2 Root 0 January 10 15:19 . drwxr-xr-x 20 Root Root 4260 Jan 10 15:07 .. crw------- 1 root root 242, 0 January 10 15:19 link control crw------- 1 root root 242, 1 January 10 15:19 my-binder Please note that the connector control device cannot be removed, as this will make the instance binderfs unusable. The link control device will be removed when a binderfs instance is not installed and all references to it are removed. binderfs: Montage of multiple instances Montage of another instance of binderfs elsewhere will create a new and separate copy from all other binderfs. This is identical to the behavior of devpts, tmpfs, and also - even if never merged in the core - kdbusfs: root@edfu: mkdir binderfs1 root@edfu: mount -t binderfs1 binderfs1 binderfs1 root@edfu: ls -al binderfs1/ Total 4 drwxr-xr-x2 root 0 January 10 15:23. drwxr-xr-x 72 ubuntu ubuntu 4096 Jan 10 15:23 .. crw------- 1 Root Root 242, January 2 10 15:23 Bind control There is no binding device in this new binderfs case because its devices are not related to those in binderfs instance on /dev/binderfs. This means that users can easily get their own personal set of binders. binderfs: The installation of bindings in user names can be installed and used to highlight new binders in user name spaces. This has the advantage that binderfs can be used in unprivileged containers or any custom solution for a sandbox based on space ubuntu@edfu:'$unshare-user--map-root-mount-root@edfu: mkdir binderfs-userns root@edfu:-t binder-user-root@edfu: BFS binary used here is a compiled program on top of root@edfu: ./bfs binderfs-userns/binder-control-user-binder 242, minor 4, and name my-user-binder root@edfu: ls-al binderfs-userns/ Total 4 drwxr-xr-x 2 Root Root 0 January 10 15:34 . drwxr-xr-x 73 Root Root 4096 Jan 10 15:32 .. crw------- 1 Root Root 242, 3 January 10 15:34 bind control crw------- 1 root root 242, 4 January 10 15:36 my-user-binding kernel Binds Bindets binderfs patchset merges upstream and will be available when Linux 5.0 gets released. There are several outstanding patches that are currently waiting in Greg's tree (cf. binderfs: remove the wrong kern_mount () call () and binderfs: make each binderfs mount a new instancechar-misc-linus) and some others are in line for a 5.1 merger window. But overall he seems to be in decent shape. Form.