Design and Implementation of Doors API for Linux

Nikita Yurievich Danilov

NikitaDanilov@Yahoo.COM

Design and Implementation of Doors API for Linux

by Nikita Yurievich Danilov

This document describes design and implementation of LDoors: Doors API in the Linux kernel. Important feature of LDoors is that door invocations are served without creation of new threads, thus hoping to reach better scalability.

Table of Contents

List of Tables

Chapter 1. Introduction

This document describes design and implementation of LDoors: Doors API in the Linux kernel. Doors are announced entry-points of fast intra-machine RPC first introduced in Solaris.

In Solaris process creates door by `door_create()` call. This call returns file descriptor, associated with STREAM. This file descriptor is usually then attached (through `fattach()`) to the file-system name-space so that other processes can open it. Any process holding door file descriptor invokes door by `door_call()` passing arguments to it. As result of this call code associated with door by the `door_create()` will be executed with given arguments in the address space of door server process (one that called `door_create()`). Invocation is finished by calling `door_return()` and control returns to the client process. Further details of Solaris' API can be found at docs.sun.com (http://docs.sun.com) web site.

To implement Doors API Solaris kernel maintains a pool of threads sleeping in door server process address space and serves door invocations by dispatching them to threads from the pool.

This interface has several advantages over traditional RPC:

- doors are located within normal file-system name space, **ls**, **mv** and **rm** work with them;
- doors are subject to normal permission checks;
- kernel can choose to implement some optimizations like use of memory re-mappings in stead of copying for argument passing or direct hand-off scheduling;
- as kernel and door server process cooperate (through `door_server_create()`) in maintaining thread pool for serving door invocations, they can balance use of precious kernel resources required to support large number of concurrent clients.

That said, it seems reasonable to implement Doors API for the Linux. Described implementation has following goals:

- be compatible with Solaris API at user-level, but not necessary at the sys-calls level;

- be scalable both CPU-wise and with respect to the number of clients;
- don't waste kernel resources especially threads;
- only require localized changes in kernel code, preferably in a form of loadable kernel module.

Chapter 2. Design

This chapter describes main features of LDoors.

2.1. Representation

In Solaris door is represented by STREAM file descriptor, created by `door_create()` and later attached to the file-system. LDoors represents door as character device with specific major number (currently 60). This way, LDoors can be implemented as loadable device driver without interfering with the rest of the kernel. Another possible solutions are:

- implement `door_create()` as new sys-call creating new struct file, populating its struct file_operations and stuffing it into caller's *`current->files->fd`* array.

  This leaves a problem of communicating this file descriptor to other processes through some form of `fattach()`. While something equivalent to `fattach()` would be useful by itself, implementation of such feature requires interaction with constantly changing VFS code.

- Implement special new file-system type doorfs, existing only in RAM like proc. This only supports transient doors not surviving reboots.
- Implement proper support for doors as new file type (in addition to regular files, directories, symlinks, pipes, etc.) in all file-system types.

It should be noted that implementing doors as character devices requires door creator process to have `CAP_MKNOD` capability. One can think of either

- using of `capset()` sys-call to add necessary capability to individual processes or process groups;
- modifying `vfs_mknod()` to allow less restrictive conditions for door creation;
- creating of "boot-strap" device `/dev/door` serving as entry point for dedicated door creation server.

In Linux minor and major numbers fully specify device, but we don't want to be limited to 255 doors in a system. Minor number of door doesn't matter. In stead doors, possibly all sharing the same major and minor number, are identified by dentry structure associated with file structure created when the door is opened for the first time.

2.2. Invocation

Think what would be result of hundreds of clients accessing door concurrently in implementation where each invocation is served in separate thread/process taken from the pool. For each client there will be two processes: one is thread serving invocation and another is client itself sleeping until invocation finishes. In particular entering and leaving a door require process switch.

This overhead (which can be bottleneck on the highly loaded system) can be avoided by serving door invocation by the client itself. More specifically: Linux process (or equivalently task) is represented in kernel by an instance of task_struct structure described in `/usr/include/linux/sched.h`.

Process can be thought of as thread (schedulable entity), address space and other attributes. To serve door invocation kernel replaces address space related fields of caller's task_struct with ones of door server. From this moment on client shares address space with server. Old values of replaced fields are stored in the kernel. When server finishes door invocation, old values are restored and client returns into its original address space. Fields replaced are:

- *`task_struct.sigpending`*
- *`task_struct.addr_limit`*
- *`task_struct.exec_domain`*
- *`task_struct.mm`*
- *`task_struct.active_mm`*
- *`task_struct.binfmt`*
- *`task_struct.personality`*
- *`task_struct.euid`* /* only when door is suid */

- *task_struct.egid* /* only when door is sgid */
- *task_struct.sig*
- *task_struct.blocked*
- *task_struct.pending*
- *task_struct.sas_ss_sp*
- *task_struct.sas_ss_size*
- *task_struct.notifier*
- *task_struct.notifier_data*
- *task_struct.notifier_mask*

This poses a problem: if all clients run in the same address space how can kernel decide where to store arguments passed? In traditional implementation each invocation is served in separate thread with distinct stack. Task of managing invocation stacks is better to be done at user-level than in kernel. LDoors communicates this information from user-level by means on simple "boostrap-protocol".

When server process first sets door up, it passes to kernel description of door arguments, address of entry point function, exit notification signal and a number of "boostrap stacks". Each boostrap stack is just area in virtual memory of server process. When client invokes door it is switches into server address space as described above. Arguments passed by client as part of invocation are stored in kernel space. Then, while client process is still in kernel, return address that was stored on the kernel stack when client made invocation sys-call is replaced with address of entry function (address in server address space) and old return address is saved. Then kernel looks for free boostrap stack, stores appropriate stack pointer as stack pointer that will be used on return from the sys-call, again saves old stack pointer and marks bootstrap stack as used.

Now client process ultimately returns from the invocation sys-call (which actually is `ioctl()`). But because of task_struct and registers manipulations just described it'll find itself running (in user mode) in server address space and executing code of door entry function using one of boostrap stacks. Door entry function should do the follows:

- allocate working stack where door invocation will be served. This can be done in a way user mode choose: `malloc()`, `mmap()`, or preallocated pool of stacks;
- switch to working stack (for example by `longjmp()`);
- select some specific "cookie value";
- allocate a space for invocation arguments;
- make "arguments passing" call, giving to it cookie and pointer to allocated argument space.

On "arguments passing" call kernel marks appropriate boostrap stack as free, stores cookie value, copies previously stored invocation arguments into user space and returns.

Now, client process is running in server address space on valid stack with arguments passed. *Uh!* It's time to do real work. When door invocation is about to exit it makes "door exit" call. On this call kernel switches caller back into saved client address space, restores program counter and stack pointer and, before exiting sends to *server* process "exit notification" signal (if configured during door setup), passing cookie in siginfo_t. This information can be used by server process to do cleanup, reclaim working stack etc.

All this complexity is not for waste. It achieves important goals.

- No new thread is created to serve door invocation. Each thread (== process in Linux) occupies at least 8K of kernel space (task_struct + kernel stack), lurks in hash-tables, queues etc.
- And therefore door invocation doesn't require process switch (going through `switch()`), only context switch (replacing user-registers, address space fields in task_struct, flushing TLB, i386's [GL]DT etc.). This can be termed in-place direct hand-off scheduling, because execution continue in context known in advance ("hand-off") without `switch()` ("direct") and without even creation on new thread ("in-place").

2.3. Manipulation

After door is opened by normal `open()` call all further access to it is through special `ioctl()` commands and, of course, ultimate explicit or implicit `close()`.

Table 2-1. Doors `ioctl()` commands

Name	Synopsis	Description
DOOR_INIT	int **ioctl (int *doorfd, DOOR_INIT, door_setup *setup);***	initialize door: set up number and type of arguments passed by clients, signal notification, bootstrap stacks etc. Called by server process to set door up.
DOOR_EXIT	int **ioctl (int *doorfd, DOOR_EXIT, int return_code);***	terminate active door invocation for the current process and return into caller VM with given return code. This is caller by server code when door execution is completed.
DOOR_ENTER	int **ioctl (int *doorfd, DOOR_ENTER, door_arg_data *args);***	start door invocation. Pass arguments to the server. This is called by client.
DOOR_ARGS	int **ioctl (int *doorfd, DOOR_ARGS, door_arg_pass *args);***	called by server code after thread switched from the bootstrap stack to the per-invocation stack. Kernel fills passed structure with arguments provided by the client.

Chapter 3. Implementation

To Be Done