TCM Userspace Design

Design

TCM is another name for LIO, an in-kernel iSCSI target (server). Existing TCM targets run in the kernel. TCMU (TCM in Userspace) allows userspace programs to be written which act as iSCSI targets. This document describes the design.

The existing kernel provides modules for different SCSI transport protocols. TCM also modularizes the data storage. There are existing modules for file, block device, RAM or using another SCSI device as storage. These are called "backstores" or "storage engines". These built-in modules are implemented entirely as kernel code.

Background

In addition to modularizing the transport protocol used for carrying SCSI commands ("fabrics"), the Linux kernel target, LIO, also modularizes the actual data storage as well. These are referred to as "backstores" or "storage engines". The target comes with backstores that allow a file, a block device, RAM, or another SCSI device to be used for the local storage needed for the exported SCSI LUN. Like the rest of LIO, these are implemented entirely as kernel code.

These backstores cover the most common use cases, but not all. One new use case that other non-kernel target solutions, such as tgt, are able to support is using Gluster's GLFS or Ceph's RBD as a backstore. The target then serves as a translator, allowing initiators to store data in these non-traditional networked storage systems, while still only using standard protocols themselves.

If the target is a userspace process, supporting these is easy. tgt, for example, needs only a small adapter module for each, because the modules just use the available userspace libraries for RBD and GLFS.

Adding support for these backstores in LIO is considerably more difficult, because LIO is entirely kernel code. Instead of undertaking the significant work to port the GLFS or RBD APIs and protocols to the kernel, another approach is to create a userspace pass-through backstore for LIO, "TCMU".

Benefits

In addition to allowing relatively easy support for RBD and GLFS, TCMU will also allow easier development of new backstores. TCMU combines with the LIO loopback fabric to become something similar to FUSE (Filesystem in Userspace), but at the SCSI layer instead of the filesystem layer. A SUSE, if you will.

The disadvantage is there are more distinct components to configure, and potentially to malfunction. This is unavoidable, but hopefully not fatal if we're careful to keep things as simple as possible.

Design constraints

- Good performance: high throughput, low latency
- Cleanly handle if userspace:
    1. never attaches
    2. hangs
    3. dies
    4. misbehaves
- Allow future flexibility in user & kernel implementations
- Be reasonably memory-efficient
- Simple to configure & run
- Simple to write a userspace backend

Implementation overview

The core of the TCMU interface is a memory region that is shared between kernel and userspace. Within this region is: a control area (mailbox); a lockless producer/consumer circular buffer for commands to be passed up, and status returned; and an in/out data buffer area.

TCMU uses the pre-existing UIO subsystem. UIO allows device driver development in userspace, and this is conceptually very close to the TCMU use case, except instead of a physical device, TCMU implements a memory-mapped layout designed for SCSI commands. Using UIO also benefits TCMU by handling device introspection (e.g. a way for userspace to determine how large the shared region is) and signaling mechanisms in both directions.

There are no embedded pointers in the memory region. Everything is expressed as an offset from the region's starting address. This allows the ring to still work if the user process dies and is restarted with the region mapped at a different virtual address.

See target_core_user.h for the struct definitions.

The Mailbox

The mailbox is always at the start of the shared memory region, and contains a version, details about the starting offset and size of the command ring, and head and tail pointers to be used by the kernel and userspace (respectively) to put commands on the ring, and indicate when the commands are completed.

version - 1 (userspace should abort if otherwise)

flags:

- TCMU_MAILBOX_FLAG_CAP_OOOC:
  indicates out-of-order completion is supported. See "The Command Ring" for details.

cmdr_off
: The offset of the start of the command ring from the start of the memory region, to account for the mailbox size.

cmdr_size
: The size of the command ring. This does *not* need to be a power of two.

cmd_head
: Modified by the kernel to indicate when a command has been placed on the ring.

cmd_tail
: Modified by userspace to indicate when it has completed processing of a command.

The Command Ring

Commands are placed on the ring by the kernel incrementing mailbox.cmd_head by the size of the command, modulo cmdr_size, and then signaling userspace via uio_event_notify(). Once the command is completed, userspace updates mailbox.cmd_tail in the same way and signals the kernel via a 4-byte write(). When cmd_head equals cmd_tail, the ring is empty -- no commands are currently waiting to be processed by userspace.

TCMU commands are 8-byte aligned. They start with a common header containing "len_op", a 32-bit value that stores the length, as well as the opcode in the lowest unused bits. It also contains cmd_id and flags fields for setting by the kernel (kflags) and userspace (uflags).

Currently only two opcodes are defined, TCMU_OP_CMD and TCMU_OP_PAD.

When the opcode is CMD, the entry in the command ring is a struct tcmu_cmd_entry. Userspace finds the SCSI CDB (Command Data Block) via tcmu_cmd_entry.req.cdb_off. This is an offset from the start of the overall shared memory region, not the entry. The data in/out buffers are accessible via tht req.iov[] array. iov_cnt contains the number of entries in iov[] needed to describe either the Data-In or Data-Out buffers. For bidirectional commands, iov_cnt specifies how many iovec entries cover the Data-Out area, and iov_bidi_cnt specifies how many iovec entries immediately after that in iov[] cover the Data-In area. Just like other fields, iov.iov_base is an offset from the start of the region.

When completing a command, userspace sets rsp.scsi_status, and rsp.sense_buffer if necessary. Userspace then increments mailbox.cmd_tail by entry.hdr.length (mod cmdr_size) and signals the kernel via the UIO method, a 4-byte write to the file descriptor.

If TCMU_MAILBOX_FLAG_CAP_OOOC is set for mailbox->flags, kernel is capable of handling out-of-order completions. In this case, userspace can handle command in different order other than original. Since kernel would still process the commands in the same order it appeared in the command ring, userspace need to update the cmd->id when completing the command(a.k.a steal the original command's entry).

When the opcode is PAD, userspace only updates cmd_tail as above -- it's a no-op. (The kernel inserts PAD entries to ensure each CMD entry is contiguous within the command ring.)

More opcodes may be added in the future. If userspace encounters an opcode it does not handle, it must set UNKNOWN_OP bit (bit 0) in hdr.uflags, update cmd_tail, and proceed with processing additional commands, if any.

The Data Area

This is shared-memory space after the command ring. The organization of this area is not defined in the TCMU interface, and userspace should access only the parts referenced by pending iovs.

Device Discovery

Other devices may be using UIO besides TCMU. Unrelated user processes may also be handling different sets of TCMU devices. TCMU userspace processes must find their devices by scanning sysfs class/uio/uio*/name. For TCMU devices, these names will be of the format:

```
tcm-user/<hba_num>/<device_name>/<subtype>/<path>
```

where "tcm-user" is common for all TCMU-backed UIO devices. <hba_num> and <device_name> allow userspace to find the device's path in the kernel target's configfs tree. Assuming the usual mount point, it is found at:

```
/sys/kernel/config/target/core/user_<hba_num>/<device_name>
```

This location contains attributes such as "hw_block_size", that userspace needs to know for correct operation.

<subtype> will be a userspace-process-unique string to identify the TCMU device as expecting to be backed by a certain handler, and <path> will be an additional handler-specific string for the user process to configure the device, if needed. The name cannot contain ':', due to LIO limitations.

For all devices so discovered, the user handler opens /dev/uioX and calls mmap():

```
mmap(NULL, size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)
```

where size must be equal to the value read from /sys/class/uio/uioX/maps/map0/size.

Device Events

If a new device is added or removed, a notification will be broadcast over netlink, using a generic netlink family name of "TCM-USER" and a multicast group named "config". This will include the UIO name as described in the previous section, as well as the UIO minor number. This should allow userspace to identify both the UIO device and the LIO device, so that after determining the device is supported (based on subtype) it can take the appropriate action.

Other contingencies

Userspace handler process never attaches:

- TCMU will post commands, and then abort them after a timeout period (30 seconds.)

Userspace handler process is killed:

- It is still possible to restart and re-connect to TCMU devices. Command ring is preserved. However, after the timeout period, the kernel will abort pending tasks.

Userspace handler process hangs:

- The kernel will abort pending tasks after a timeout period.

Userspace handler process is malicious:

- The process can trivially break the handling of devices it controls, but should not be able to access kernel memory outside its shared memory areas.

Writing a user pass-through handler (with example code)

A user process handing a TCMU device must support the following:

a. Discovering and configuring TCMU uio devices
b. Waiting for events on the device(s)
c. Managing the command ring: Parsing operations and commands, performing work as needed, setting response fields (scsi_status and possibly sense_buffer), updating cmd_tail, and notifying the kernel that work has been finished

First, consider instead writing a plugin for tcmu-runner. tcmu-runner implements all of this, and provides a higher-level API for plugin authors.

TCMU is designed so that multiple unrelated processes can manage TCMU devices separately. All handlers should make sure to only open their devices, based opon a known subtype string.

a. Discovering and configuring TCMU UIO devices:

```
/* error checking omitted for brevity */

int fd, dev_fd;
char buf[256];
unsigned long long map_len;
void *map;

fd = open("/sys/class/uio/uio0/name", O_RDONLY);
ret = read(fd, buf, sizeof(buf));
close(fd);
buf[ret-1] = '\0'; /* null-terminate and chop off the \n */

/* we only want uio devices whose name is a format we expect */
if (strncmp(buf, "tcm-user", 8))
  exit(-1);

/* Further checking for subtype also needed here */

fd = open(/sys/class/uio/%s/maps/map0/size, O_RDONLY);
ret = read(fd, buf, sizeof(buf));
close(fd);
str_buf[ret-1] = '\0'; /* null-terminate and chop off the \n */

map_len = strtoull(buf, NULL, 0);
```

```
dev_fd = open("/dev/uio0", O_RDWR);
map = mmap(NULL, map_len, PROT_READ|PROT_WRITE, MAP_SHARED, dev_fd, 0);


b) Waiting for events on the device(s)

while (1) {
  char buf[4];

  int ret = read(dev_fd, buf, 4); /* will block */

  handle_device_events(dev_fd, map);
}
```

c. Managing the command ring:

```
#include <linux/target_core_user.h>

int handle_device_events(int fd, void *map)
{
  struct tcmu_mailbox *mb = map;
  struct tcmu_cmd_entry *ent = (void *) mb + mb->cmdr_off + mb->cmd_tail;
  int did_some_work = 0;

  /* Process events from cmd ring until we catch up with cmd_head */
  while (ent != (void *)mb + mb->cmdr_off + mb->cmd_head) {

    if (tcmu_hdr_get_op(ent->hdr.len_op) == TCMU_OP_CMD) {
      uint8_t *cdb = (void *)mb + ent->req.cdb_off;
      bool success = true;

      /* Handle command here. */
      printf("SCSI opcode: 0x%x\n", cdb[0]);

      /* Set response fields */
      if (success)
        ent->rsp.scsi_status = SCSI_NO_SENSE;
      else {
        /* Also fill in rsp->sense_buffer here */
        ent->rsp.scsi_status = SCSI_CHECK_CONDITION;
      }
    }
    else if (tcmu_hdr_get_op(ent->hdr.len_op) != TCMU_OP_PAD) {
      /* Tell the kernel we didn't handle unknown opcodes */
      ent->hdr.uflags |= TCMU_UFLAG_UNKNOWN_OP;
    }
    else {
      /* Do nothing for PAD entries except update cmd_tail */
    }

    /* update cmd_tail */
    mb->cmd_tail = (mb->cmd_tail + tcmu_hdr_get_len(&ent->hdr)) % mb->cmdr_size;
    ent = (void *) mb + mb->cmdr_off + mb->cmd_tail;
    did_some_work = 1;
  }

  /* Notify the kernel that work has been finished */
  if (did_some_work) {
    uint32_t buf = 0;

    write(fd, &buf, 4);
  }

  return 0;
}
```

A final note

Please be careful to return codes as defined by the SCSI specifications. These are different than some values defined in the scsi/scsi.h include file. For example, CHECK CONDITION's status code is 2, not 1.