EXPORTING SCSI DEVICES THROUGH CTL

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Suraj Ponugoti

FALL
2016

EXPORTING SCSI DEVICES THROUGH CTL

A Project

by

Suraj Ponugoti

Approved by:

_____, Committee Chair
Dr. Jinsong Ouyang

_____, Second Reader
Dr. Yuan Cheng

_____
Date

Student: Suraj Ponugoti

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator _____
Dr. Ying Jin Date

Department of Computer Science

Abstract

of

EXPORTING SCSI DEVICES THROUGH CTL

by

Suraj Ponugoti

In current FreeBSD version, there are no applications/ports available to export Small Computer System Interface (SCSI) devices through iSCSI. This project is about exporting physical SCSI devices on server side to initiator side (Client Device) over iSCSI through CAM Target Layer (CTL) as an actual SCSI target. CTL is a Disk, CD-ROM and Tape Drive emulation system. The backing storage for these luns is Ramdisk/file backend.

These luns are visible through camsim and iSCSI frontend port. Camsim frontend port allows us using CTL without any physical hardware. iSCSI frontend port allows us to export CTL luns via iSCSI to the initiator. In CTL for representing actual SCSI devices I used a new Logical Unit Number (LUN) called Passthrough. All the luns in CTL use either Block/file or Ramdisk as their backend, but Passthrough luns does not require both of those, because these Passthrough luns acts as dummy luns which redirects all the IO's (CCB) to actual SCSI device.

I have written a new backend driver for these luns, which redirects all the IO's to actual SCSI device. iSCSI frontend submits the IO using the ctl_queue, and CTL need to redirect all the SCSI commands coming over iSCSI for SCSI device luns to CAM layer. To pass these IO's (CCB) to CAM layer, CTL needs to pass them to any peripheral of that SCSI device. But CAM layer doesn't have any peripheral driver which accepts IO's(CCB) from CTL, so I have written a new peripheral driver.

_____, Committee Chair
Dr. Jinsong Ouyang

_____
Date

ACKNOWLEDGMENTS

I would like to thank Dr. Jinsong Ouyang for taking up the role of my project advisor. His feedback and guidance helped me see the real purpose of my master's project.

In addition, I would like to thank Dr. Yuan Cheng for his willingness to serve on the committee. In addition, I would like to thank the entire faculty and staff of the Department of Computer Science at California State University, Sacramento.

TABLE OF CONTENTS

Page

`

LIST OF FIGURES

Figures Pages

Chapter1

INTRODUCTION

1.1 Overview

In current FreeBSD kernel, CTL (CAM Target Layer) is a Disk, CD-ROM and Processor device emulated system. This project is about exporting server side physical SCSI devices to initiator side (Client Device) over iSCSI through CTL as an actual SCSI target.

I have written a new CTL backend driver for representing actual SCSI devices in CTL and a new peripheral driver which converts all the CTL commands to CAM commands.

I made few changes to userland code for representing SCSI devices. These commands are passed to actual device and status is returned to client.

1.2 Features and Functionalities

The user can send SCSI commands via CTL to Actual SCSI devices using CTL administrative utility(Ctladm), iSCSI and CAM control administrative utility (Cam Control).

With this project, I can virtually Mount SCSI device to the client file system via CTL-iSCSI, and format SCSI devices via ISCSI through CTL.

I can also use CTL SCSI device as a backup device on client side, and copy data from CTL SCSI device to a local device on client side.

Chapter 2

OVERVIEW OF THE TECHNOLOGIES USED

2.1 Basic Terminology

SCSI (Small Computer System Interface): It is a fast bus that can connect lots of devices to a computer at the same time, including hard drive, scanners, CD-ROM, printers and tape drives [1].

For example: optical drives like CD-ROM or tape communicates with the system through SCSI tunneled over SATA port.

SCSI Commands: Computer as a client requests storages devices to perform some operation using SCSI commands [2].

Commands are send to devices in CDB (command descriptor block) along with opcode and other command specific parameters. Once the storage device receives the request it process it and return status back to computer (client).

CAM Layer: System Programmers faced lot of challenges to implement software in communicating with SCSI devices. So, programmers have designed CAM (Common Access Method) framework for communicating with SCSI devices [3].

CCB (CAM Control Block): CAM Layer sends SCSI and other administrative commands in the form of CCB to SCSI device frontend Driver. Frontend driver converts those into CDB and sends it to actual device for execution [3].

Peripheral Drivers: Peripheral devices acts as an interface between Operating System and CAM Layer. Applications running on system use general services like read, writes and ioctl to access these devices [3].

These syscalls are passed to peripheral drivers and these drivers send appropriate SCSI command to the device using CAM layer.

CTL (CAM Target Layer): It is a SCSI devices emulation subsystem, which can be served to other machines using Fiber Channel or iSCSI. It executes SCSI commands, just like a hard drive, or a DVD drive, or a tape, would [4].

Currently CTL supports disk, processor and CD-ROM emulation. CTL provides block/file or Ramdisk as a backend for the emulated devices (luns).

iSCSI: It runs on top of the Transport Control Protocol (TCP) and allows SCSI commands to be send over network. It works at the block device level [5].

LUN (Logical Unit Number): It is used by SCSI or other storage area network protocols for addressing device.

Ctladm: It is a CAM Target Layer (CTL) control utility designed to provide access to CTL. It provides a way to send SCSI commands to the CTL layer [6].

Camcontrol: This utility provides a way for users to access and control the FreeBSD CAM subsystem [7].

iscsictl: This utility is used to configure iSCSI initiator [8].

2.2 Technologies Used

2.2.1 FreeBSD: FreeBSD is an open source operating system used on different processors. It mainly focuses on speed and stability. It is derived from a version of UNIX developed at the University of California, Berkeley [9]. Due to its advance security, extensive file systems, easier customization and network stack, system programmers prefer using FreeBSD over Linux in servers. In FreeBSD, there are more than 20,000 ports and packages available to install applications. Source code of FreeBSD kernel is available in many version control systems. Companies like WhatsApp, Netflix uses FreeBSD for their servers.

2.2.2 GDB debugger: GNU Project debugger allows us to see what is going on ‘inside’ another program while it executes or to see the reasons behind any system crash [10]. GDB debugger helps us to examine what has happened when a line of code is executed. GDB debugger gives us a stack trace and line of code that caused crash. The program which is being debugged can be written in C, C++, Objective-C (and many other languages).

2.2.3 dTrace: dTrace is performance analysis and troubleshooting tool used in various operating systems [11]. It is used as a debugging tools both in system level and user level.

Chapter 3

OVERVIEW OF IMPLEMENTATION

3.1 Application Design

In current FreeBSD version CTL (CAM Target Layer) is a Hard Disk, CD-ROM and Tape Drive emulated system. The backend for these luns can be either Block or Ramdisk. If I choose backend as a block, then I need to specify the drive or file name for storing the data and for Ramdisk I need to specify the size of the Ram drive for the lun. These luns are accessible to outside world through frontend ports.

We mainly have two frontend ports for accessing luns.

- Internal Frontend Port - camsim
- External Frontend Port - iSCSI

By enabling Internal frontend port(camsim) - CTL attaches these luns to CAM layer. camsim frontend port allows us using CTL luns without any physical hardware. Users can issue SCSI commands to device using pass(4)/da(4) devices [4].

When users access CTL luns, CAM sends all the CCB (Cam control block) to internal frontend port(camsim) and the port redirects the actual commands to respective luns.

By enabling external frontend port(iSCSI)- It allows us to export CTL luns via iSCSI to the initiator. On the initiator side device nodes for these devices appear in /dev/. These devices should be mounted before using it.

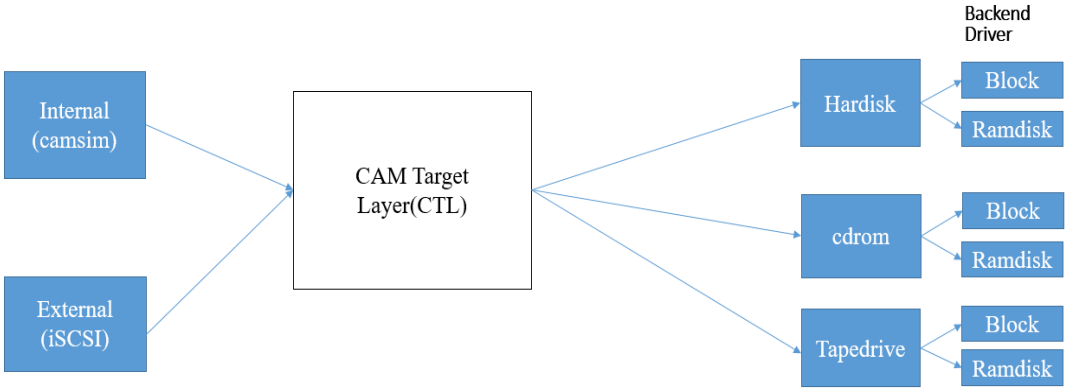

Figure 1. CTL Architecture Diagram

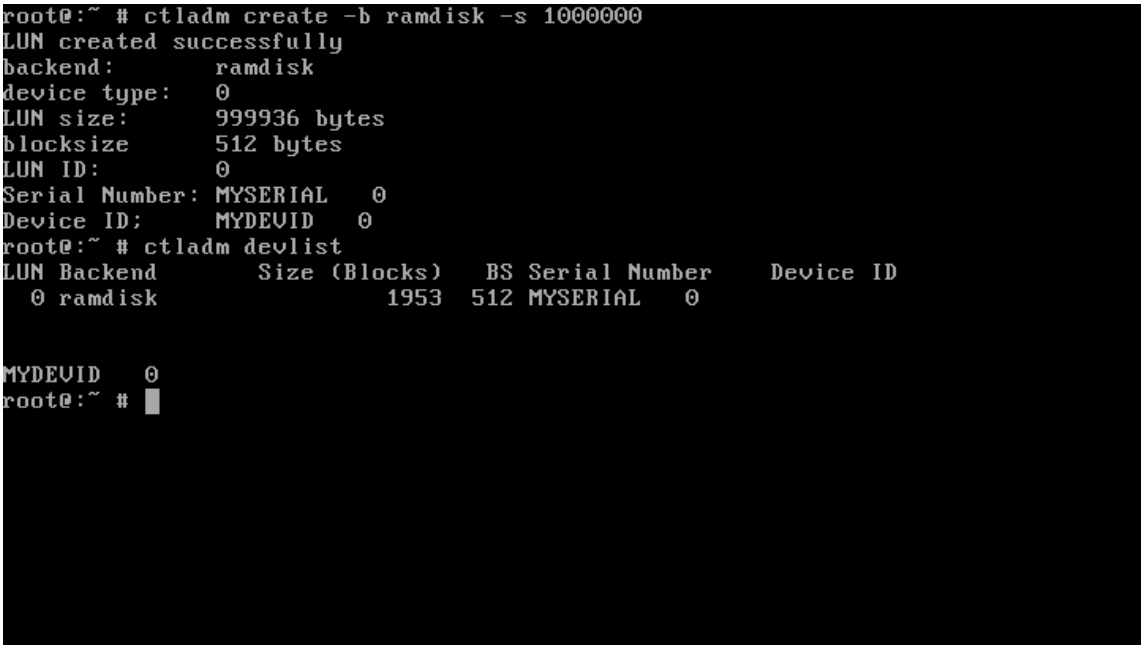

Figure 2. Creating Emulated Drive in CTL.

In the above screen shot I have created a new emulated disk lun whose backend is ramdisk. Ctladm acts as an administrative utility where we can create, remove luns or pass SCSI commands to CTL.

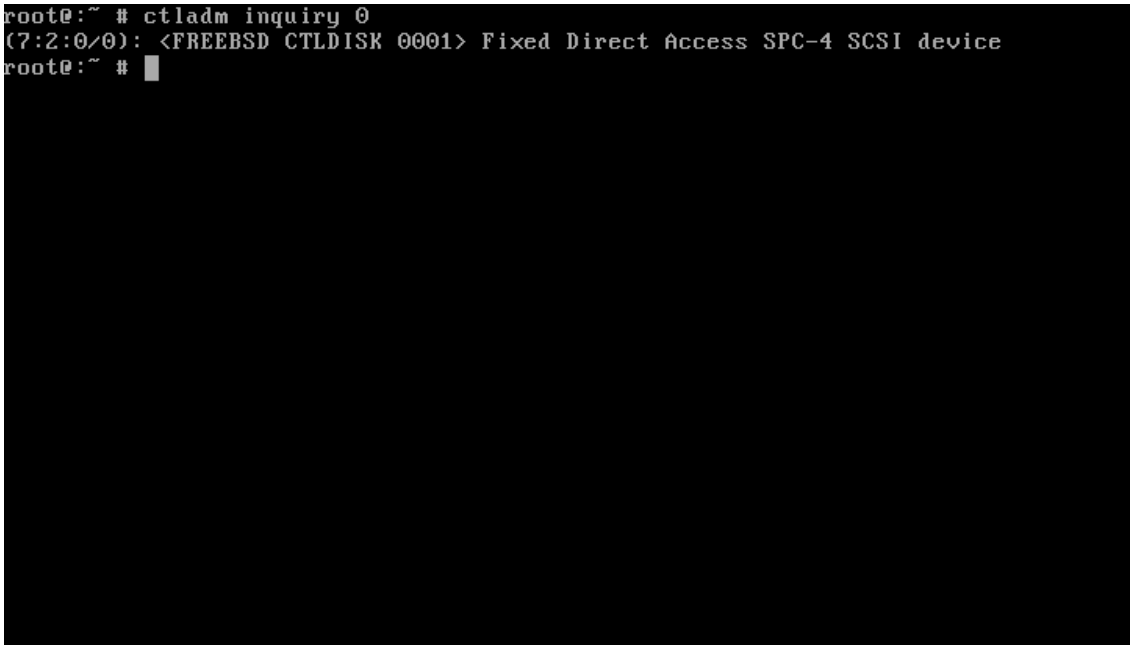

Figure 3. Calling inquiry command on the CTL lun

In the above screen shot I have passed SCSI inquiry command to CTL emulated disk lun.

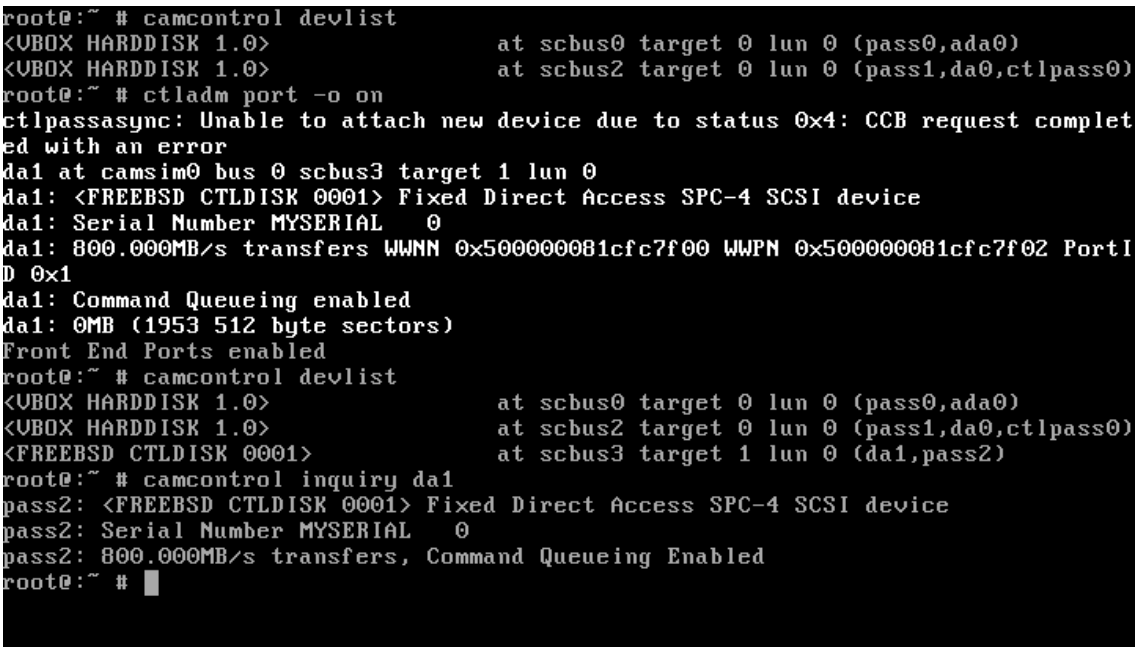

Figure 4. Camcontrol devlist command before and after attaching CTL luns to CAM

Camcontrol devlist command list all the devices attached to the CAM layer. By using ‘ctladm port -o on’ command all the CTL luns are attached to CAM layer. The screenshot above shows us the before and after ctladm port was enabled. [5]

In the above screenshot, CTL lun is represented using device node ‘da1’. So, all the commands which are given to device are passed to CTL for execution.

Similarly, we can export CTL luns using iSCSI frontend port to any initiator(client). To export luns we need to represent which luns we want to export in ctl.conf configuration file.

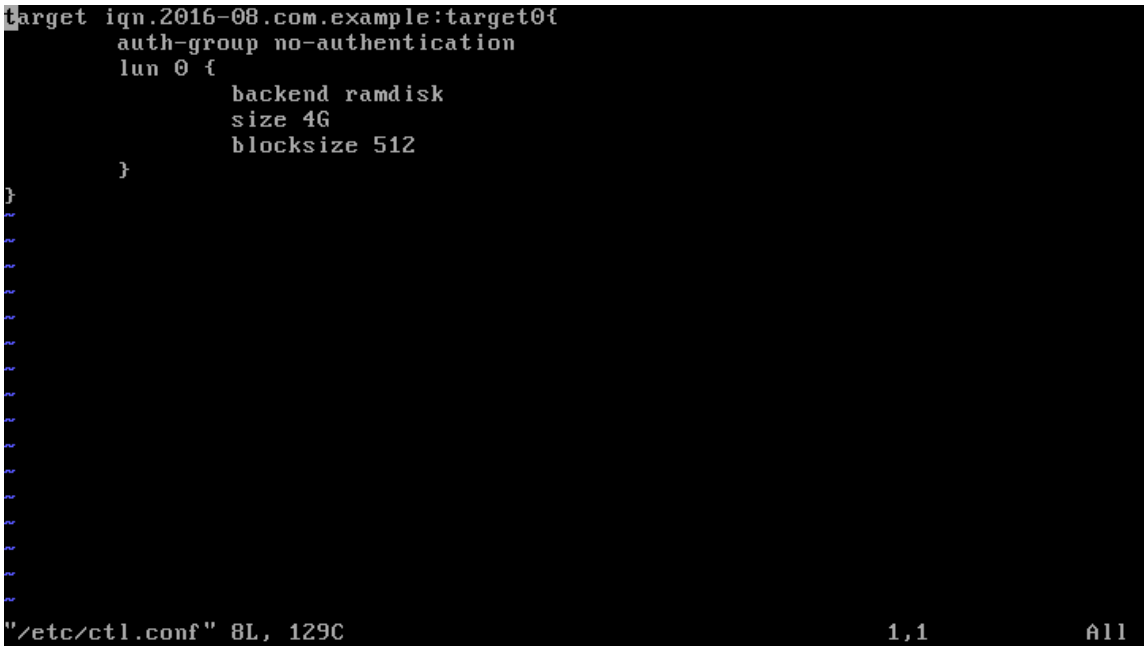

Figure 5. ctl.conf configuration file

In ctl.conf file I need to define the luns which I want to export. In the above screenshot, I have defined a new target name (logical name) for the system. When ctld is started it parses ctl.conf and sees whether CTL (kernel) having luns mentioned in ctl.conf or not. If it doesn't have the luns mentioned, then ctld automatically creates those luns for us [12].

On the initiator side (client):

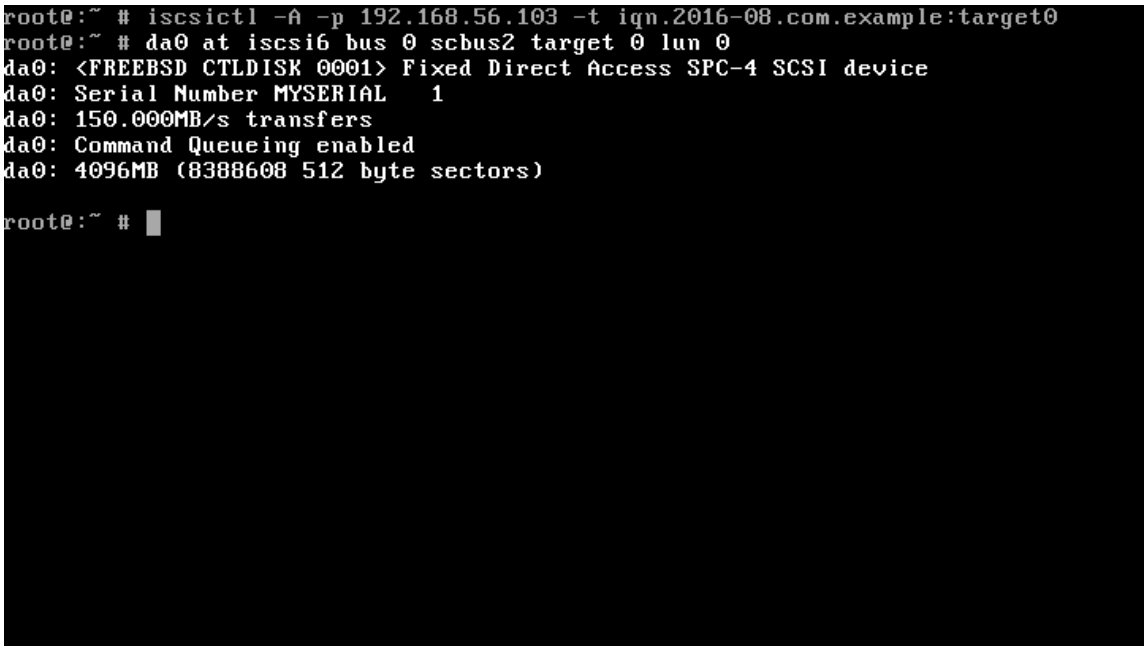

Figure 6. Connecting CTL luns using iSCSI from initiator side

Initiator needs to connect host by using its IP address and target name. Once operating systems detects the device it assigns a device node for the device.

This project is about exporting actual SCSI devices through CTL front end ports.

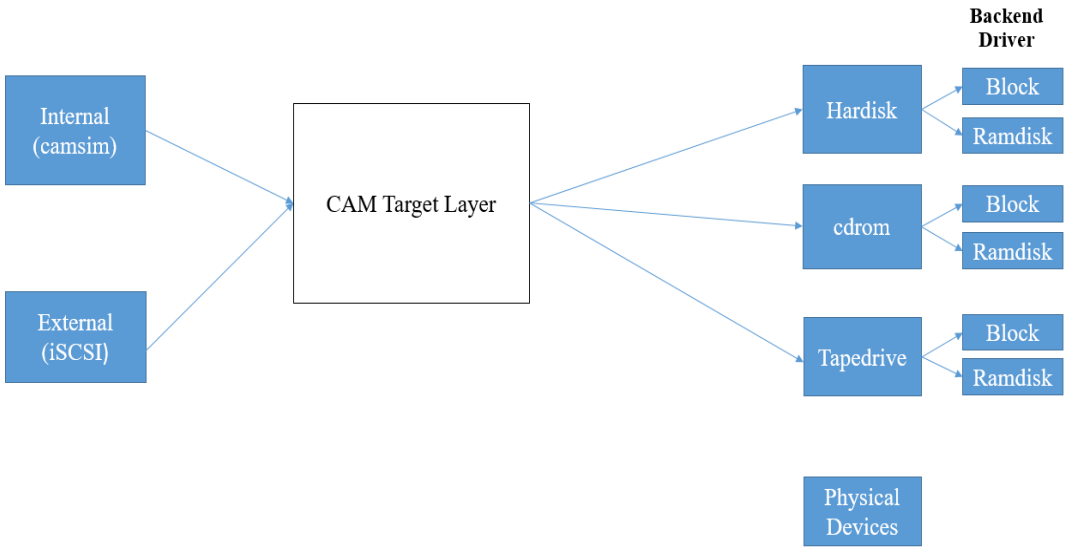

Figure 7. Abstract view of exporting SCSI devices through CTL

In CTL for representing actual SCSI devices I have created a new lun type called Passthrough. All the commands coming to the CTL luns are operated on a block or a ramdisk. But here CTL need to send these commands to actual devices. I have written a new backend driver for redirecting the commands to actual devices.

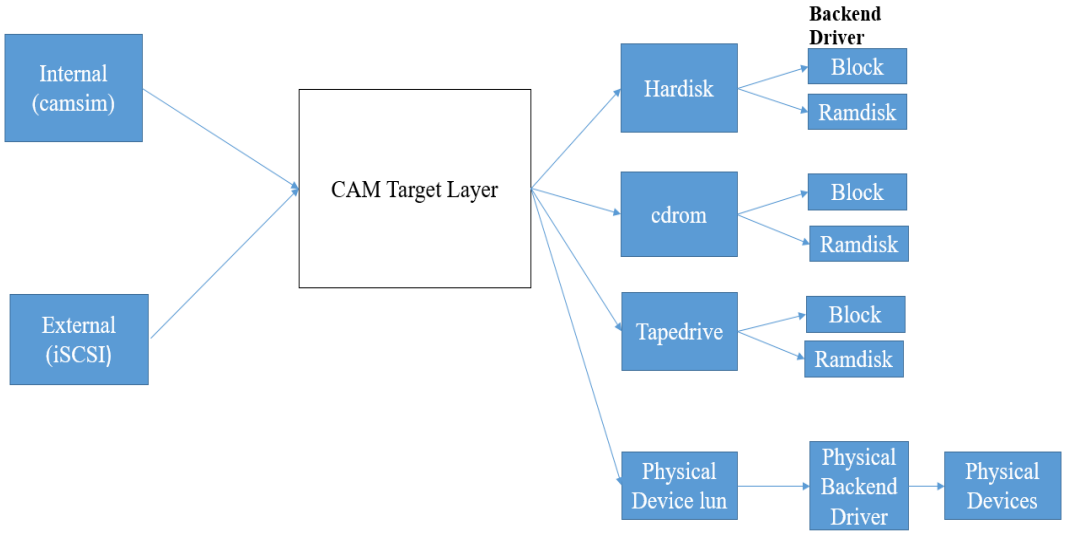

Figure 8. Adding Passthrough luns to CTL

Generally, applications running on the system use operating system services like read, write, ioctl to access peripheral devices without knowing actual SCSI commands. Peripheral drivers convert user related actions to SCSI commands and passed to devices. But there is no peripheral driver which accepts IO request (SCSI commands) from CTL.

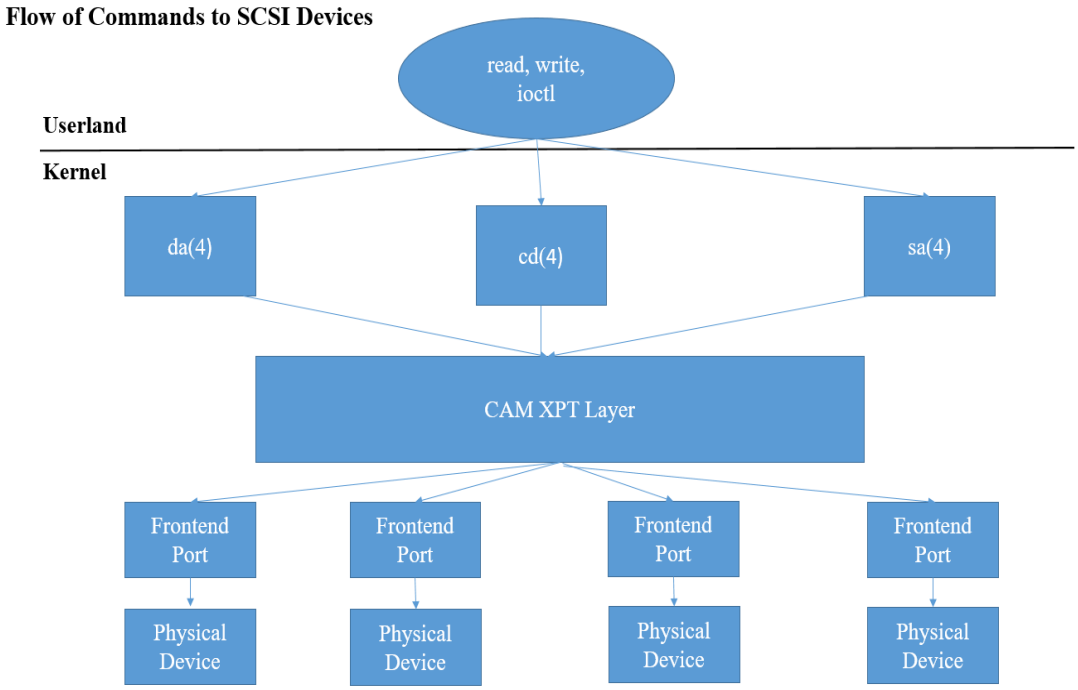

Figure 9. Flow of commands to SCSI devices in FreeBSD

Above screen shoot describes the basic flow of SCSI commands in FreeBSD Operating System.

I have written a new peripheral driver which accepts IO request (SCSI commands) from CTL and creates a new IO request based on CTL IO request and sends the request to actual device.

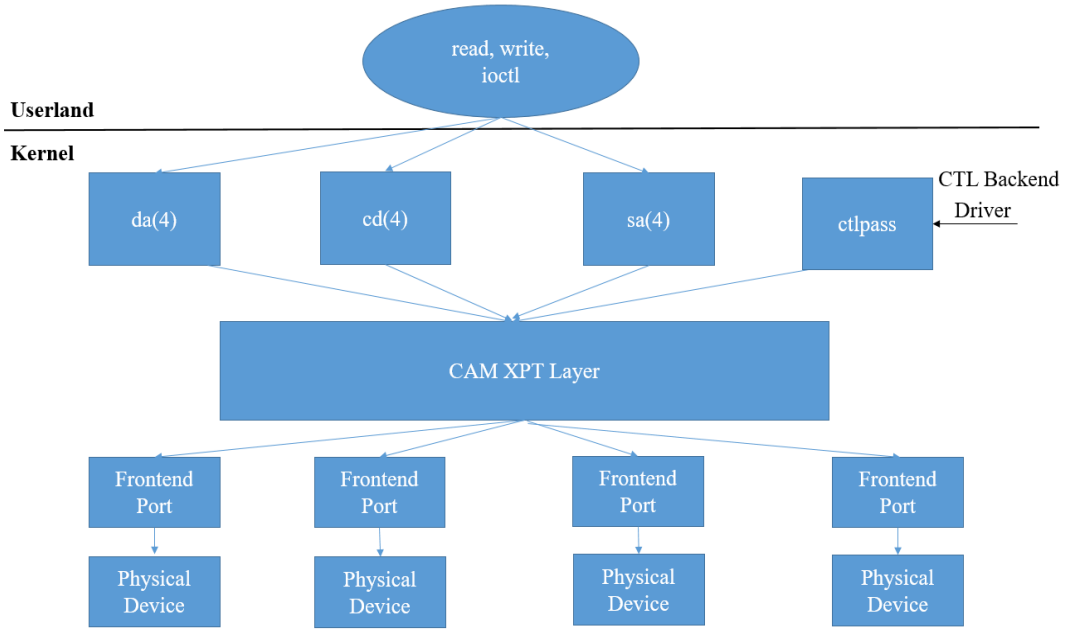

Figure 10. Adding new peripheral driver to CAM Layer

Chapter 4

DESIGN AND IMPLEMENTATION OF APPLICATION

4.1 Representing actual SCSI devices in CTL

I first started creating a new lun called Passthrough for representing SCSI devices in CTL. All the SCSI commands on these luns should be redirected to actual device. I have written a new backend driver for Passthrough luns.

This driver handles

- Creating new Passthrough luns
- Deleting Passthrough luns
- Updating Passthrough luns
- Shutting down Passthrough luns
- Copying of data from User land to kernel data buffer or vice versa

4.1.1 Creating Passthrough luns

Passthrough luns are created in the same way the other CTL emulated luns are created, but these luns act as dummy luns.

For creating Passthrough luns user need to pass the bus, path and lun id of the SCSI device as input.

Passthrough backend driver checks whether is there any lun already created for that device. If not, then it creates a new lun for this device and adds this lun in the lunlist.

Creating of Passthrough luns can happen in two ways.

- Using Ctladm create command
- Using ctld (which handles iSCSI connections for CTL)

I made few changes to ctladm create command so the user can pass bus, path and lun id as arguments with the command.

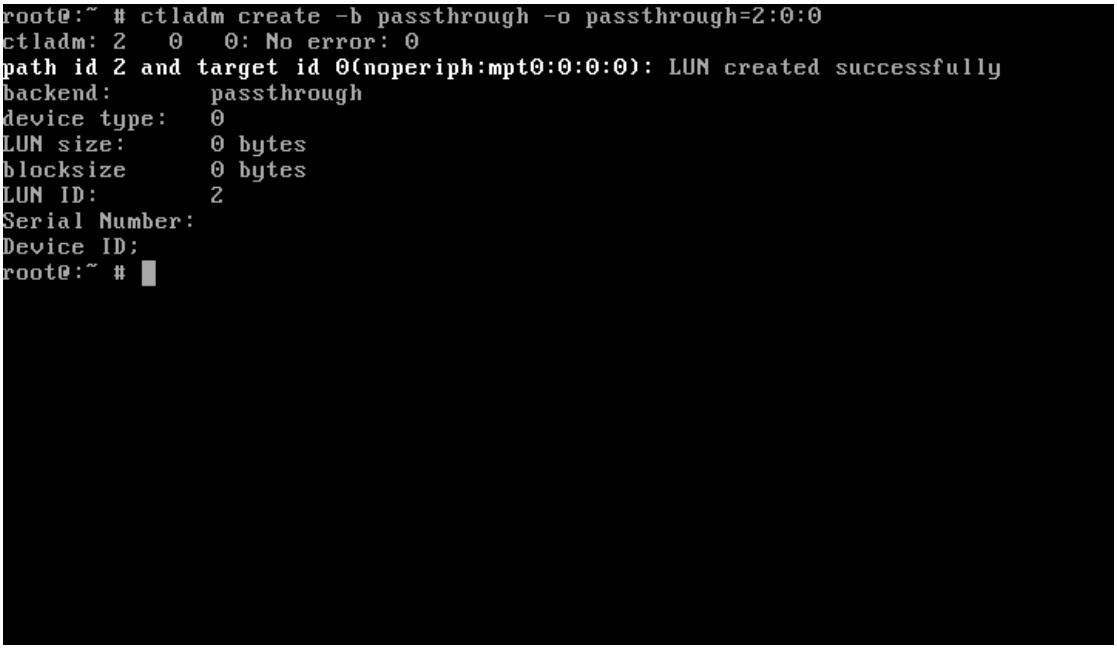

Figure 11. Creating a new Passthrough lun using ctladm command

In the screenshot above I have created a new Passthrough lun for the SCSI device residing on bus, path and lun id-2,0,0.

Made changes to ctl.conf configuration file code, so that appropriate arguments required for creating Passthrough luns is passed to the Passthrough backend driver.

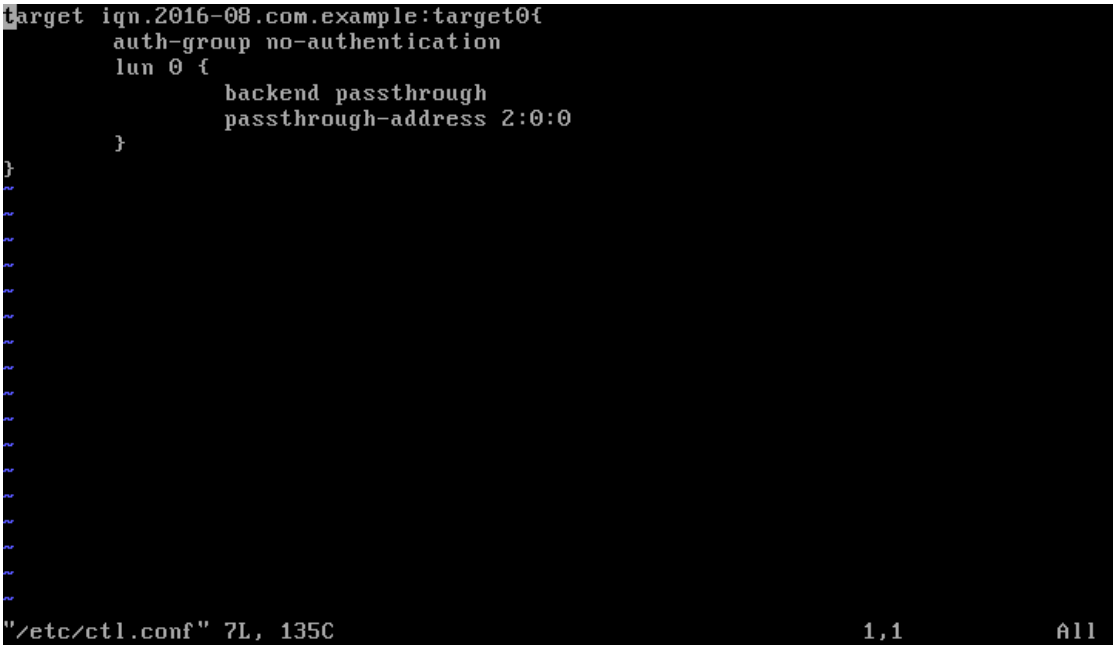

Figure 12. ctl.conf configuration file for exporting Passthrough luns

In the above attached screenshot, I am passing bus, path and lun id using a new argument called Passthrough-address. When ctld gets started it parses ctl.conf file and passes these arguments ctl Passthrough backend driver.

Code snippet and explanation:

For Passthrough lun creation user need to pass all the parameters to backend driver using lun_req structure variable. Backend driver checks whether system has any SCSI device for given path, target and lun id. If there is a SCSI device, then driver also checks whether it

has any LUN in CTL which is representing this SCSI device. Once it checks all these conditions, then it passes those parameters to passthrough create function for lun creation.

```
case CTL_LUNREQ_CREATE:{
        u_int path_id;
        u_int target_id;
        u_int32_t lun_id;
        struct cam_path *path = NULL;
        struct cam_periph *periph=NULL;
        struct ctl_lun_create_params *params = NULL;
        char *name ="ctlpass";
        params = &lun_req->reqdata.create;
        path_id = params->scbus;
        target_id = params->target;
        lun_id = params->lun_num;
if(xpt_create_path(&path, NULL, path_id, target_id, lun_id) == CAM_REQ_CMP)
{
        if(path == NULL)
                printf("path is NULL");
        xpt_print_path(path);
        xpt_path_lock(path);
 periph = cam_periph_find(path,name);
 xpt_path_unlock(path);
 xpt_free_path(path);
 retval = ctl_backend_passthrough_create(periph,lun_req);
}
        lun_req->status = CTL_LUN_OK;
        break;
 }
```

For creating CTL Passthrough lun, backend driver need to configure details of the LUN. LUN details mainly includes path, target, lun id, block size, peripheral pointer pointing to the actual SCSI device, max logical block address and backend type. Passthrough backend

driver calls ctl_add_lun function of CTL for adding this Passthrough lun to CTL. Once CTL successfully creates a new lun it updates its lun list and returns 0.

```
int ctl_backend_passthrough_create(struct cam_periph *periph,struct ctl_lun_req *lun_req)
{
	struct ctl_lun_create_params *params;
	struct ctl_be_passthrough_softc *softc = &rd_softc;
	struct ctl_be_passthrough_lun *be_lun;
	struct ctl_be_lun *cbe_lun;
	char *value;
	char num_thread_str[16];
	int num_threads=0 , tmp_num_threads=0;

	char tmpstr[32];
	int retval;
	retval = 0;
	params = &lun_req->reqdata.create;
	if(periph==NULL)
		printf("periph is NULL");
	be_lun = malloc(sizeof(*be_lun), M_PASSTHROUGH, M_ZERO | M_WAITOK);
	if(be_lun == NULL)
	{
		goto bailout_error;
	}
	STAILQ_INIT(&be_lun->cbe_lun.options);
	cbe_lun = &be_lun->cbe_lun;
	cbe_lun->be_lun = be_lun;
	be_lun->params = lun_req->reqdata.create;
	be_lun->cbe_lun.lun_type = T_PASSTHROUGH;
	be_lun->softc=softc;
	be_lun->periph = periph;
	be_lun->flags = CTL_BE_PASSTHROUGH_LUN_UNCONFIGURED;
	cbe_lun->flags =0 ;
	ctl_init_opts(&cbe_lun->options, lun_req->num_be_args, lun_req->kern_be_args);
	cbe_lun->scbus = params->scbus;
```

```
        cbe_lun->target = params->target;
        cbe_lun->lun = params->lun_num;
        value = ctl_get_opt(&cbe_lun->options, "num_threads");
    if (value != NULL) {
            tmp_num_threads = strtol(value, NULL, 0);
    if (tmp_num_threads < 1) {
                    snprintf(lun_req->error_str, sizeof(lun_req->error_str),
                            "invalid number of threads %s",
                            num_thread_str);
                    goto bailout_error;
            }
            num_threads = tmp_num_threads;
    }
        be_lun->num_threads = num_threads;
        be_lun->cbe_lun.maxlba=0xffffffff;
        cbe_lun->blocksize=512;
        be_lun->size_bytes = 0;
        be_lun->size_blocks =0;
        cbe_lun->flags |=CTL_LUN_FLAG_UNMAP;
        be_lun->cbe_lun.flags = CTL_LUN_FLAG_PRIMARY;
        be_lun->cbe_lun.be_lun = be_lun;
        be_lun->cbe_lun.req_lun_id=0;

        be_lun->cbe_lun.lun_shutdown = ctl_backend_passthrough_lun_shutdown;
        be_lun->cbe_lun.lun_config_status = ctl_backend_passthrough_lun_config_status;
        be_lun->cbe_lun.be = &ctl_be_passthrough_driver;
                    snprintf(tmpstr, sizeof(tmpstr), "MYSERIAL%4d",
                                softc->num_luns);
                    strncpy((char *)cbe_lun->serial_num, tmpstr,
                            MIN(sizeof(cbe_lun->serial_num), sizeof(tmpstr)));

                    snprintf(tmpstr, sizeof(tmpstr), "MYDEVID%4d", softc->num_luns);
                    strncpy((char *)cbe_lun->device_id, tmpstr,
                            MIN(sizeof(cbe_lun->device_id), sizeof(tmpstr)));

                    mtx_lock(&softc->lock);
                    softc->num_luns++;
        STAILQ_INSERT_TAIL(&softc->lun_list , be_lun , links);
        mtx_unlock(&softc->lock);
        retval = ctl_add_lun(&be_lun->cbe_lun);
        if(retval!=0)
        {
                            mtx_lock(&softc->lock);
```

```
                    STAILQ_REMOVE(&softc->lun_list , be_lun ,ctl_be_passthrough_lun,
links);
                    softc->num_luns--;
                mtx_unlock(&softc->lock);
                    retval =0;
                    goto bailout_error;
        }
        mtx_lock(&softc->lock);
        be_lun->flags |= CTL_BE_PASSTHROUGH_LUN_WAITING;

while (be_lun->flags & CTL_BE_PASSTHROUGH_LUN_UNCONFIGURED) {
                    retval = msleep(be_lun, &softc->lock, PCATCH, "ctlpassthrough", 0);
                    if (retval == EINTR)
                            break;
        }
        be_lun->flags &= ~CTL_BE_PASSTHROUGH_LUN_WAITING;

if(be_lun->flags & CTL_BE_PASSTHROUGH_LUN_CONFIG_ERR){
        STAILQ_REMOVE(&softc->lun_list, be_lun ,ctl_be_passthrough_lun, links);
                    softc->num_luns--;
                    mtx_unlock(&softc->lock);
                    goto bailout_error;
        }
        else{
        params->req_lun_id = cbe_lun->lun_id;
        }
        mtx_unlock(&softc->lock);
        return (retval);
bailout_error:
        return (retval);
}
```

4.1.2 Removing Passthrough lun

Removing of luns can be done using ctladm remove command. This command calls appropriate backend driver remove method.

Figure 13. ctladm remove command for removing Passthrough lun

In the above screenshot, ctladm remove command is used to remove Passthrough lun having lun id 0.

Code snippet and explanation:

For removing passthrough luns, backend driver calls ctl_backend_passthrough remove method. This method takes two parameters, current state of passthrough backend driver(softc) and lun id. This method checks for the lun id in the lun list and disables it. If it doesn't find any lun with this id in the lun list, it updates the status of the request with CTL_LUN_ERROR and returns 0.

```
static int ctl_backend_passthrough_remove(struct ctl_be_passthrough_softc *softc,struct
ctl_lun_req *req){
struct ctl_be_passthrough_lun *be_lun;
        struct ctl_lun_rm_params *params;
        int retval;

        params = &req->reqdata.rm;
        mtx_lock(&softc->lock);
        STAILQ_FOREACH(be_lun, &softc->lun_list, links) {
                if (be_lun->cbe_lun.lun_id == params->lun_id)
                        break;
        }
        mtx_unlock(&softc->lock);
        if (be_lun == NULL) {
                snprintf(req->error_str, sizeof(req->error_str),
                         "%s: LUN %u is not managed by the passthrough backend",
                         __func__, params->lun_id);
                goto bailout_error;
        }
        retval = ctl_disable_lun(&be_lun->cbe_lun);
        if (retval != 0) {
                snprintf(req->error_str, sizeof(req->error_str),
                         "%s: error %d returned from ctl_disable_lun() for "
                         "LUN %d", __func__, retval, params->lun_id);
                goto bailout_error;
        }
mtx_lock(&softc->lock);
        be_lun->flags |= CTL_BE_PASSTHROUGH_LUN_WAITING;
        mtx_unlock(&softc->lock);
        retval = ctl_invalidate_lun(&be_lun->cbe_lun);
        if (retval != 0) {
                snprintf(req->error_str, sizeof(req->error_str),
                         "%s: error %d returned from ctl_invalidate_lun() for "
                         "LUN %d", __func__, retval, params->lun_id);
                mtx_lock(&softc->lock);
                be_lun->flags &= ~CTL_BE_PASSTHROUGH_LUN_WAITING;
                mtx_unlock(&softc->lock);
                goto bailout_error;
        }
        mtx_lock(&softc->lock);
        be_lun->flags &= ~CTL_BE_PASSTHROUGH_LUN_WAITING;
        if (retval == 0) {
                STAILQ_REMOVE(&softc->lun_list, be_lun, ctl_be_passthrough_lun,
```

```
                                links);
                softc->num_luns--;
        }

        mtx_unlock(&softc->lock);

        if (retval == 0) {
                ctl_free_opts(&be_lun->cbe_lun.options);
                free(be_lun, M_PASSTHROUGH);
        }

        req->status = CTL_LUN_OK;
        return (retval);

bailout_error:
        req->status = CTL_LUN_ERROR;
        return (0);
}
```

4.1.3 Updating Passthrough lun

Updating of luns can be done using ctladm modify command. This command calls appropriate modify method of the backend driver.

Code snippet and explanation:

This function is used for changing the size of the lun. It takes modified size as the parameter.

```
 static int ctl_backend_passthrough_modify(struct ctl_be_passthrough_softc *softc,struct
ctl_lun_req *req){
struct ctl_be_passthrough_lun *be_lun;
        struct ctl_be_lun *cbe_lun;
        struct ctl_lun_modify_params *params;
        char *value
```

```
	uint32_t blocksize;
	params = &req->reqdata.modify;
	mtx_lock(&softc->lock);
	STAILQ_FOREACH(be_lun, &softc->lun_list, links) {
		if (be_lun->cbe_lun.lun_id == params->lun_id)
			break;
	}
	mtx_unlock(&softc->lock);
	if (be_lun == NULL) {
		snprintf(req->error_str, sizeof(req->error_str),
			 "%s: LUN %u is not managed by the passthrough backend",
			 __func__, params->lun_id);
		goto bailout_error;
	}
	cbe_lun = &be_lun->cbe_lun;
	if (params->lun_size_bytes != 0)
		be_lun->params.lun_size_bytes = params->lun_size_bytes;
	ctl_update_opts(&cbe_lun->options, req->num_be_args, req->kern_be_args);
	blocksize = be_lun->cbe_lun.blocksize;
	if (be_lun->params.lun_size_bytes < blocksize) {
		snprintf(req->error_str, sizeof(req->error_str),
			"%s: LUN size %ju < blocksize %u", __func__,
			be_lun->params.lun_size_bytes, blocksize);
			goto bailout_error;
	}
	be_lun->size_blocks = be_lun->params.lun_size_bytes / blocksize;
	be_lun->size_bytes = be_lun->size_blocks * blocksize;
	be_lun->cbe_lun.maxlba = be_lun->size_blocks - 1;
	ctl_lun_capacity_changed(&be_lun->cbe_lun);
	params->lun_size_bytes = be_lun->size_bytes;
	req->status = CTL_LUN_OK;
	return (0);
bailout_error:
	req->status = CTL_LUN_ERROR;
	return (0);
}
```

4.1.4 Creating kernel data buffer for copying data from User space

For various SCSI commands, we need to move data from kernel buffer to user land buffer or vice versa. Passthrough backend driver helps in moving data in.

Code Snippet and explanation:

This function helps in copying data from user data buffer to kernel data buffer. This function creates a kernel buffer of 128kb and calls ctl_datamove function for copying data from user data buffer.

```
 static void ctl_backend_passthrough_continue(union ctl_io *io)
{

	int len;
	len = io->io_hdr.ctl_private[CTL_PRIV_BACKEND].integer;
	io->scsiio.kern_data_ptr = malloc(128000,M_PASSTHROUGH,M_WAITOK);
	io->scsiio.be_move_done = ctl_backend_passthrough_move_done;
	io->scsiio.kern_data_resid = 0;
	io->scsiio.kern_data_len = 128000;
	io->scsiio.kern_sg_entries = 0;
	io->io_hdr.flags |= CTL_FLAG_ALLOCATED;
	io->io_hdr.ctl_private[CTL_PRIV_BACKEND].integer -= len;
	ctl_datamove(io);

}
```

4.2 Using ctladm utility for passing SCSI commands to device

Ctladm is a control utility used for controlling CTL layer. Ctladm provides user to pass SCSI commands to CTL layer.

Before exporting Passthrough luns through frontend port I worked on passing SCSI commands to actual devices through Passthrough luns.

Passthrough backend driver should redirect all the commands to peripheral driver, but there is no peripheral driver which accepts commands from CTL.

I have written a new peripheral driver called ctlpass which accepts commands from CTL and pass it to actual device for execution.

Ctlpass peripheral driver registers with the CAM Layer. CAM transport layer notifies registered peripheral driver arrival, departure and other asynchronous events and provides round robin prioritized scheduler for CCB.

4.2.1 Registering SCSI device with ctlpass peripheral driver

CAM Transport layer notifies ctlpass peripheral driver on arrival of new SCSI device. Each ctlpass peripheral device has four queues. Each queue has unique functionality. All the CCB request are stored in incoming queue. Once the CCB is ready to be processed it stored in ready queue. If the CCB is invalid, then it is stored in aborted queue and all the processed CCB are stored in done queue.

Code snippet and explanation:

Before registering a device, ctlpass peripheral driver creates queues for the device. It sends an inquiry command to the device and once it is successful ctlpass peripheral driver gets the peripheral reference of the device. Using the peripheral reference ctlpass driver register the device using appropriate unit number. Also peripheral driver needs to add a async call back with the CAM layer using xpt_register async function. This callback function notifies the driver if the device is lost.

```
static cam_status ctlpassregister(struct cam_periph *periph, void *arg)
{
	struct ctlpass_softc *softc;
	struct ccb_getdev *cgd;
	struct ccb_pathinq cpi;
	struct make_dev_args args;
	int error, no_tags;
	const char *str ="camsim";
	const char *str1;
	struct cam_sim *sim;
	sim = periph->sim;
	str1= cam_sim_name(sim);
	if(ctlstrcmp(str1, str)==0)
	{
		return(CAM_REQ_CMP_ERR);
	}
	cgd = (struct ccb_getdev *)arg;
	if (cgd == NULL) {
		printf("%s: no getdev CCB, can't register device\n", __func__);
		return(CAM_REQ_CMP_ERR);
	}
	softc = (struct ctlpass_softc *)malloc(sizeof(*softc),M_DEVBUF, M_NOWAIT);
	if (softc == NULL) {
	printf("%s: Unable to probe new device. ""Unable to allocate softc\n", __func__);
		return(CAM_REQ_CMP_ERR);
	}
	bzero(softc, sizeof(*softc));
```

```
	softc->state = CTLPASS_STATE_NORMAL;
	if (cgd->protocol == PROTO_SCSI)
		softc->pd_type = SID_TYPE(&cgd->inq_data);
	periph->softc = softc;
	softc->periph = periph;
	TAILQ_INIT(&softc->incoming_queue);
	TAILQ_INIT(&softc->active_queue);
	TAILQ_INIT(&softc->abandoned_queue);
	TAILQ_INIT(&softc->done_queue);
	snprintf(softc->zone_name, sizeof(softc->zone_name), "%s%d",
		 periph->periph_name, periph->unit_number);
	snprintf(softc->io_zone_name, sizeof(softc->io_zone_name), "%s%dIO",
		 periph->periph_name, periph->unit_number);
	softc->io_zone_size = MAXPHYS;
	knlist_init_mtx(&softc->read_select.si_note, cam_periph_mtx(periph));

	bzero(&cpi, sizeof(cpi));
	xpt_setup_ccb(&cpi.ccb_h, periph->path, CAM_PRIORITY_NORMAL);
	cpi.ccb_h.func_code = XPT_PATH_INQ;
	xpt_action((union ccb *)&cpi);

	if (cpi.maxio == 0)
		softc->maxio = DFLTPHYS;	/* traditional default */
	else if (cpi.maxio > MAXPHYS)
		softc->maxio = MAXPHYS;		/* for safety */
	else
		softc->maxio = cpi.maxio;	/* real value */
	if (cpi.hba_misc & PIM_UNMAPPED)
		softc->flags |= PASS_FLAG_UNMAPPED_CAPABLE;
cam_periph_unlock(periph);
	no_tags = (cgd->inq_data.flags & SID_CmdQue) == 0;
	softc->device_stats = devstat_new_entry("ctlpass",
				periph->unit_number, 0,
				DEVSTAT_NO_BLOCKSIZE
				| (no_tags ? DEVSTAT_NO_ORDERED_TAGS : 0),
				softc->pd_type |
				XPORT_DEVSTAT_TYPE(cpi.transport) |
				DEVSTAT_TYPE_PASS,
				DEVSTAT_PRIORITY_PASS);
if (cam_periph_acquire(periph) != CAM_REQ_CMP) {
		xpt_print(periph->path, "%s: lost periph during "
			"registration!\n", __func__);
		cam_periph_lock(periph);
```

```
                return (CAM_REQ_CMP_ERR);
        }
        /* Register the device */
        make_dev_args_init(&args);
        args.mda_devsw = &ctlpass_cdevsw;
        args.mda_unit = periph->unit_number;
        args.mda_uid = UID_ROOT;
        args.mda_gid = GID_OPERATOR;
        args.mda_mode = 0600;
        args.mda_si_drv1 = periph;
        error = make_dev_s(&args, &softc->dev, "%s%d", periph->periph_name,
           periph->unit_number);
        if (error != 0) {
                cam_periph_lock(periph);
                cam_periph_release_locked(periph);
                return (CAM_REQ_CMP_ERR);
        }
if (cam_periph_acquire(periph) != CAM_REQ_CMP) {
                xpt_print(periph->path, "%s: lost periph during "
                        "registration!\n", __func__);
                cam_periph_lock(periph);
                return (CAM_REQ_CMP_ERR);
        }

        cam_periph_lock(periph);

        TASK_INIT(&softc->add_physpath_task, /*priority*/0,
                  ctlpass_add_physpath, periph);
        taskqueue_enqueue(taskqueue_thread, &softc->add_physpath_task);
        xpt_register_async(AC_LOST_DEVICE | AC_ADVINFO_CHANGED,
                           ctlpassasync, periph, periph->path);
if (bootverbose)
                xpt_announce_periph(periph, NULL);
        return(CAM_REQ_CMP);
}
```

4.2.2 Unregistering SCSI device with ctlpass peripheral driver

When a SCSI device is lost, CAM layer notifies to appropriate peripheral driver. Ctlpass peripheral calls ctlpassoninvalidate function when the device is lost. All the queues related

to the SCSI device need to be cleaned up and peripheral driver notifies the devfs that the device is lost. Peripheral driver reduces the device count and cleans up all the things related to the device.

Ctlpassoninvalidate deregisters async call backs from the CAM layer and calls ctlpassdevgonecb function for cleaningup its state.

Ctlpassdevgonecb reduces the count for this device and calls ctlpassrejectios for removing all the IO's present in all the queues. Ctlpasscleanup checks whether all the queues are cleanedup.

Code snippet and explanation:

```
static void ctlpassrejectios(struct cam_periph *periph)
{
	struct ctlpass_io_req *io_req, *io_req2;
	struct ctlpass_softc *softc;

	softc = (struct ctlpass_softc *)periph->softc;
	TAILQ_FOREACH_SAFE(io_req, &softc->done_queue, links, io_req2) {
		TAILQ_REMOVE(&softc->done_queue, io_req, links);
		ctlpassiocleanup(softc, io_req);
		uma_zfree(softc->pass_zone, io_req);
	}
	TAILQ_FOREACH_SAFE(io_req, &softc->incoming_queue, links, io_req2) {
		TAILQ_REMOVE(&softc->incoming_queue, io_req, links);
		ctlpassiocleanup(softc, io_req);
		uma_zfree(softc->pass_zone, io_req);
	}
	TAILQ_FOREACH_SAFE(io_req, &softc->active_queue, links, io_req2) {
```

```
                TAILQ_REMOVE(&softc->active_queue, io_req, links);
                io_req->flags |= PASS_IO_ABANDONED;
                TAILQ_INSERT_TAIL(&softc->abandoned_queue, io_req, links);
        }
if ((!TAILQ_EMPTY(&softc->abandoned_queue))
         && ((softc->flags & PASS_FLAG_ABANDONED_REF_SET) == 0)) {
                cam_periph_doacquire(periph);
                softc->flags |= PASS_FLAG_ABANDONED_REF_SET;
        }
}

static void ctlpassdevgonecb(void *arg)
{
        struct cam_periph *periph;
        struct mtx *mtx;
        struct ctlpass_softc *softc;
        int i;

        periph = (struct cam_periph *)arg;
        mtx = cam_periph_mtx(periph);
        mtx_lock(mtx);

        softc = (struct ctlpass_softc *)periph->softc;
        KASSERT(softc->open_count >= 0, ("Negative open count %d",
                softc->open_count));
        for (i = 0; i < softc->open_count; i++)
                cam_periph_release_locked(periph);

        softc->open_count = 0;
        cam_periph_release_locked(periph);
        ctlpassrejectios(periph);
         mtx_unlock(mtx);
}

static void ctlpassoninvalidate(struct cam_periph *periph)
{
        struct ctlpass_softc *softc;

        softc = (struct ctlpass_softc *)periph->softc;
        xpt_register_async(0, ctlpassasync, periph, periph->path);
        softc->flags |= PASS_FLAG_INVALID;
        destroy_dev_sched_cb(softc->dev, ctlpassdevgonecb, periph);
}
```

```
static void ctlpasscleanup(struct cam_periph *periph)
{
	struct ctlpass_softc *softc;
	softc = (struct ctlpass_softc *)periph->softc;
	cam_periph_assert(periph, MA_OWNED);
	KASSERT(TAILQ_EMPTY(&softc->active_queue),
		("%s called when there are commands on the active queue!\n",
		__func__));
	KASSERT(TAILQ_EMPTY(&softc->abandoned_queue),
		("%s called when there are commands on the abandoned queue!\n",
		__func__));
	KASSERT(TAILQ_EMPTY(&softc->incoming_queue),
		("%s called when there are commands on the incoming queue!\n",
		__func__));
	KASSERT(TAILQ_EMPTY(&softc->done_queue),
		("%s called when there are commands on the done queue!\n",
		__func__));

	devstat_remove_entry(softc->device_stats);
	cam_periph_unlock(periph);


	taskqueue_drain(taskqueue_thread, &softc->add_physpath_task);
	cam_periph_lock(periph);
	free(softc, M_DEVBUF);
}
```

Once ctlpass peripheral driver accepts IO's (commands) form CTL. For every request, it creates a new CCB and manually copies CTL IO contents to CCB and queues it to incoming queue. Ctlpass pass peripheral driver request CCB from CAM layer for that device. and then copies CCB data from queued CCB to Device specific CCB and passed it down the line through CAM Layer for execution.

Once the CCB is executed ctlpass peripheral driver ctlpassdone will get called. Again, the data and status is copied to CTL CCB and passed it to CTL. CTL passes the status to appropriate CTL frontend port.

In the Screen shot attached below we could see SCSI devices are represented by ctlpass peripheral device node along with other device nodes.

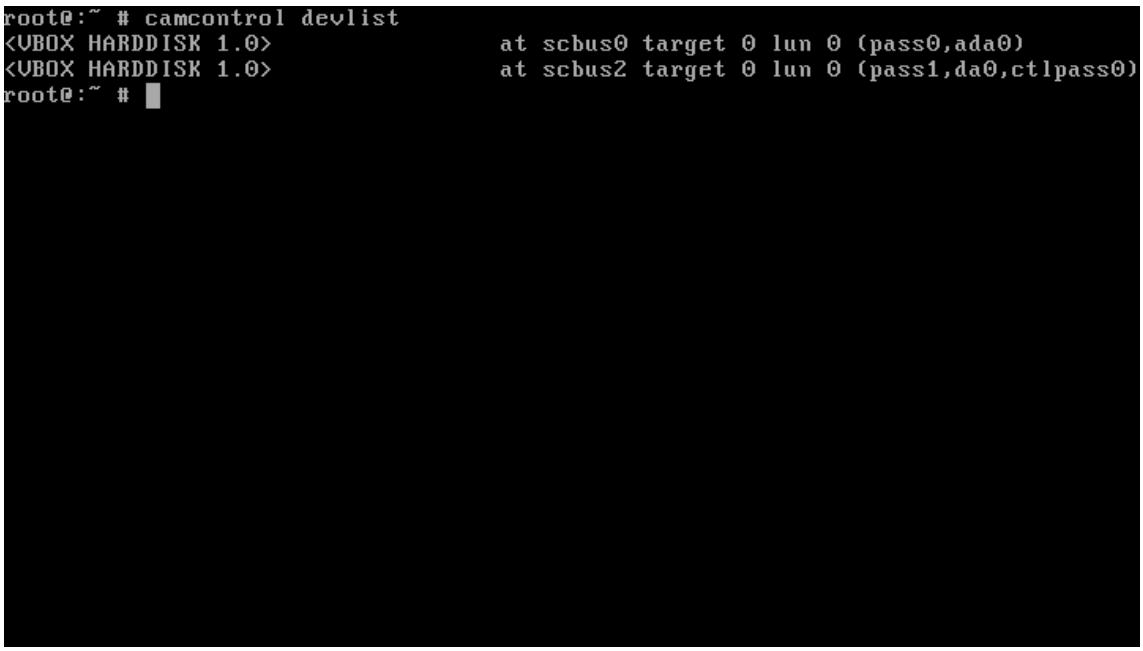

Figure 14. SCSI devices are represented using ctlpass peripheral node

4.3 Passing SCSI commands to Passthrough lun using ctladm

Below are few SCSI commands passed to actual SCSI devices through Passthrough luns using ctladm utility.

4.3.1 Passing Inquiry command

Figure 15. Calling SCSI inquiry command on Passthrough lun

Sending inquiry command to the device: Inquiry command displays some of the inquiry returned data to the user.

Code Snippet and explanation:

In ctlpass peripheral driver, once it gets CTL INQUIRY command I copied all the contents manually to CAM CCB and passed it down the CAM layer for execution.

```
        struct scsi_inquiry *scsi_cmd;
        scsi_cmd = (struct scsi_inquiry *)&csio->cdb_io.cdb_bytes;
        bzero(scsi_cmd, sizeof(*scsi_cmd));
        csio->cdb_len = sizeof(*scsi_cmd);
        csio->ccb_h.flags = CAM_DIR_IN;
        csio->sense_len = SSD_FULL_SIZE;
         csio->data_ptr =malloc(io->scsiio.kern_data_len,M_SCSIPASSTHROUGH,
M_WAITOK);
        csio->dxfer_len = io->scsiio.kern_data_len;
        break;
```

4.3.2 Passing Write Command

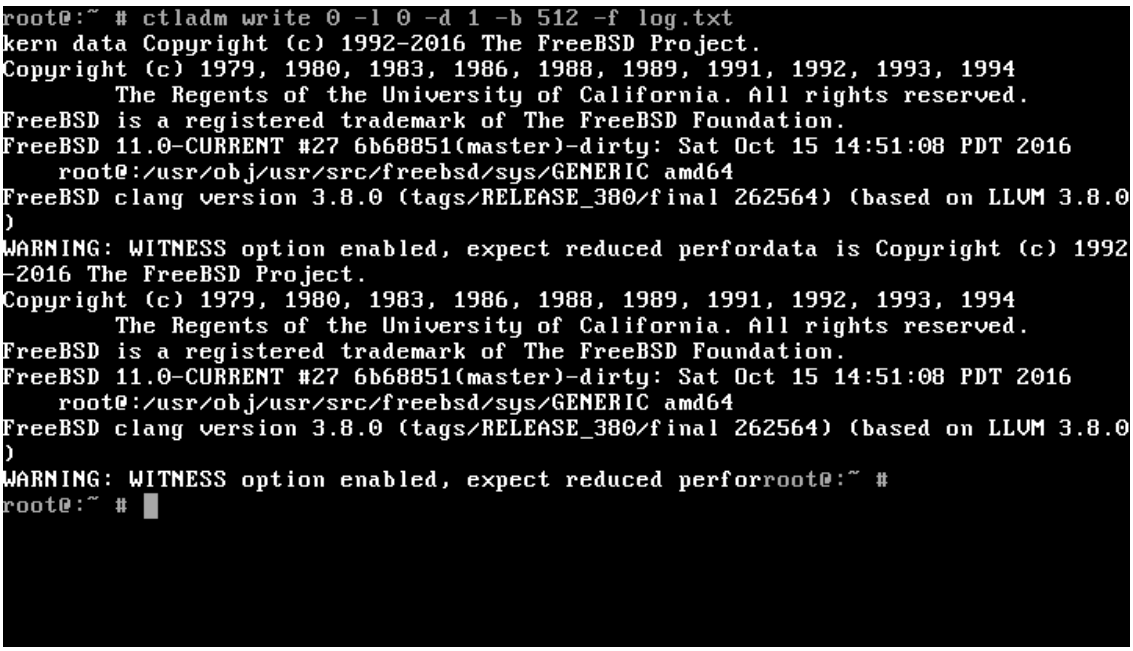

Figure 16. Calling SCSI write command on Passthrough lun

Sending Write command by reading data from a file and writing data to the file by specifying logical block address, data length and block size.

Code Snippet and explanation:

In ctlpass peripheral driver, once it gets CTL WRITE command I copied all the contents manually to CAM CCB and passed it down the CAM layer for execution. I have implemented various WRITE commands based on the buffer size.

```
case WRITE_6:{

        struct scsi_rw_6 *scsi_cmd;
        scsi_cmd = (struct scsi_rw_6 *)&csio->cdb_io.cdb_bytes;
        csio->cdb_len = sizeof(*scsi_cmd);
        io->scsiio.kern_rel_offset =0 ;
        bzero(scsi_cmd, sizeof(*scsi_cmd));

        csio->ccb_h.flags = CAM_DIR_OUT;
        csio->sense_len = SSD_FULL_SIZE;
        csio->data_ptr = (uint8_t *)malloc(io-
>scsiio.kern_data_len,M_SCSIPASSTHROUGH, M_WAITOK);
        csio->dxfer_len = io->scsiio.kern_data_len;

        memcpy(csio->data_ptr,io->scsiio.kern_data_ptr,io->scsiio.kern_data_len);
        break;
}
```

4.3.3 Passing read command

Figure 17. Calling SCSI read command on Passthrough lun

Sending Read command to the SCSI device through CTL. We need to pass appropriate logical block address to read, data length and specify the file name for copying data of the read command.

Code Snippet and explanation:

In ctlpass peripheral driver, once it gets CTL READ command I copied all the contents manually to CAM CCB and passed it down the CAM layer for execution.

```
case READ_6:{
                struct scsi_rw_6 *scsi_cmd;
                scsi_cmd = (struct scsi_rw_6 *)&csio->cdb_io.cdb_bytes;
                bzero(scsi_cmd, sizeof(*scsi_cmd));
                csio->cdb_len = sizeof(*scsi_cmd);
                io->scsiio.kern_rel_offset =0 ;
                csio->ccb_h.flags = CAM_DIR_IN;
                csio->sense_len = SSD_FULL_SIZE;
                io->scsiio.ext_data_len =128000;
                io->scsiio.kern_data_ptr =malloc(io->scsiio.ext_data_len ,M_CTL,
M_WAITOK);
                io->scsiio.kern_data_len = io->scsiio.ext_data_len;
csio->data_ptr = (uint8_t *)malloc(io>scsiio.kern_data_len,M_SCSIPASSTHROUGH,
M_WAITOK);
                csio->dxfer_len = io->scsiio.kern_data_len;
                break;
        }
```

4.3.4 Passing readcap command

Figure 18. Calling SCSI read cap command on Passthrough lun

Sending readcap command to the device through CTL. Readcap returns device size and block size.

Code Snippet and explanation:

In ctlpass peripheral driver, once it gets CTL READ CAPACITY command I copied all the contents manually to CAM CCB and passed it down the CAM layer for execution.

```
case READ_CAPACITY:{
                struct scsi_read_capacity *scsi_cmd;
                scsi_cmd = (struct scsi_read_capacity *)&csio->cdb_io.cdb_bytes;
                bzero(scsi_cmd, sizeof(*scsi_cmd));
                io->scsiio.kern_data_ptr =(uint8_t *)malloc(sizeof(struct
scsi_read_capacity_data_long),M_CTL, M_WAITOK);
                io->scsiio.kern_data_len = sizeof(struct scsi_read_capacity_data_long);
                csio->cdb_len = sizeof(*scsi_cmd);
                csio->ccb_h.flags = CAM_DIR_IN;
                csio->sense_len = SSD_FULL_SIZE;
                csio->data_ptr =(uint8_t *)malloc(sizeof(struct
scsi_read_capacity_data_long),M_SCSIPASSTHROUGH, M_WAITOK);
                csio->dxfer_len = sizeof(struct scsi_read_capacity_data_long);
                break;
        }
```

4.3.5 Passing start-stop command

Figure 19. Calling ctladm start command on Passthrough lun

Sending start command to the Device. Start command passes SCSI START STOP COMMAND with start bit set. It tells the device to execute I/O request.

Figure 20. Calling ctladm stop command on Passthrough lun

Sending stop command to the Device. Stop command passes SCSI START STOP COMMAND with start bit cleared. It tells the device to stop taking new I/O request.

Code snippet and explanation:

In ctlpass peripheral driver, once it gets CTL START_STOP command I copied all the contents manually to CAM CCB and passed it down the CAM layer for execution.

```
case START_STOP_UNIT:
                {
                        struct scsi_start_stop_unit *scsi_cmd;
                        int extra_flags=0;
                        scsi_cmd = (struct scsi_start_stop_unit *)&csio->cdb_io.cdb_bytes;
                        bzero(scsi_cmd , sizeof(*scsi_cmd));
                        csio->cdb_len = sizeof(*scsi_cmd);
                        extra_flags |= CAM_HIGH_POWER;
                      csio->ccb_h.flags = CAM_DIR_NONE|extra_flags;
                        csio->sense_len = SSD_FULL_SIZE;
                        csio->data_ptr = NULL;
                        csio->dxfer_len = 0;

                        break;
                }
```

4.4 Exporting Passthrough luns through iSCSI:

After I could pass SCSI commands from CTL Passthrough luns to SCSI devices, I worked on exporting these luns through frontend ports.

For exporting SCSI luns through external port, we need to define luns in ctl.conf configuration file. Ctld is a daemon which manages ctl.conf configuration file. If the luns defined in ctl.conf not created in CTL kernel then it creates new luns as necessary, and removes luns which are no longer existing in the configuration file. It also manages authentication of the request.

Figure 21. Calling ctladm stop command on Passthrough lun

In the attached screenshot, ctl.conf file is having one lun of type Passthrough representing SCSI device which is on bus, path and lun id- 2,0,0.

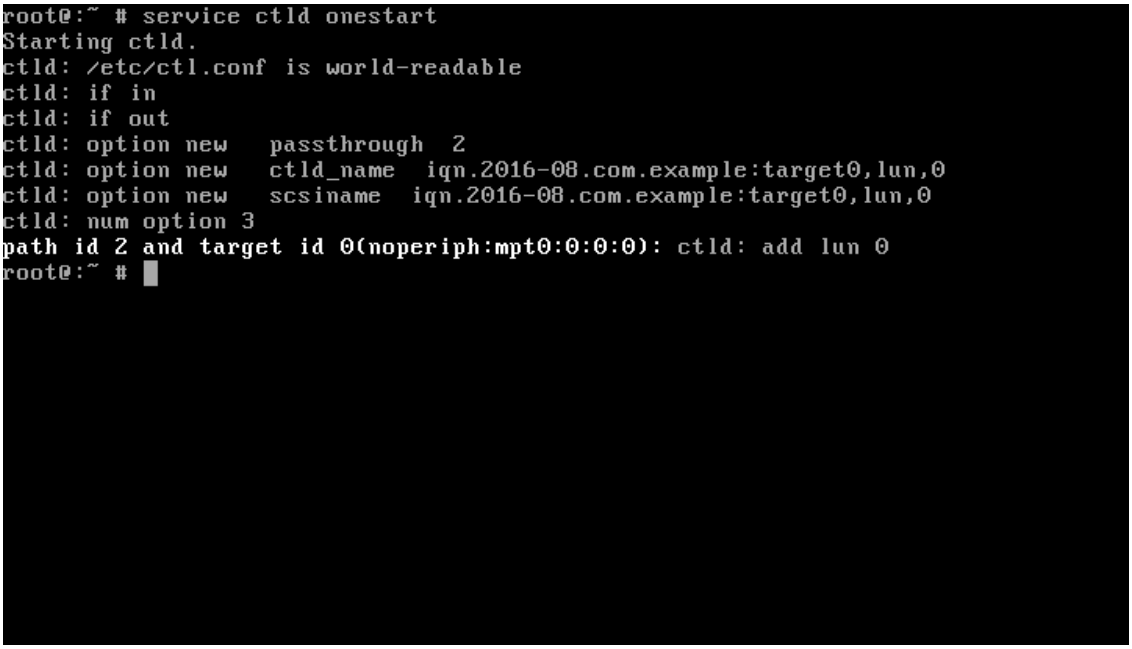

Figure 22. Calling ctld service on server side (host)

Once ctld is started any initiator having IP address and logical name of the host can access these luns.

On the initiator side, we client need to start iscsid daemon which is responsible for performing login phase of iSCSI connections [13].

Figure 23. Calling ctld service on server side (host)

Once iscsid gets started, I used iscsictl utility for configuring iSCSI initiator. Using iscsictl client can connect with the host using provided host IP address and Target name (logical name) of the host.

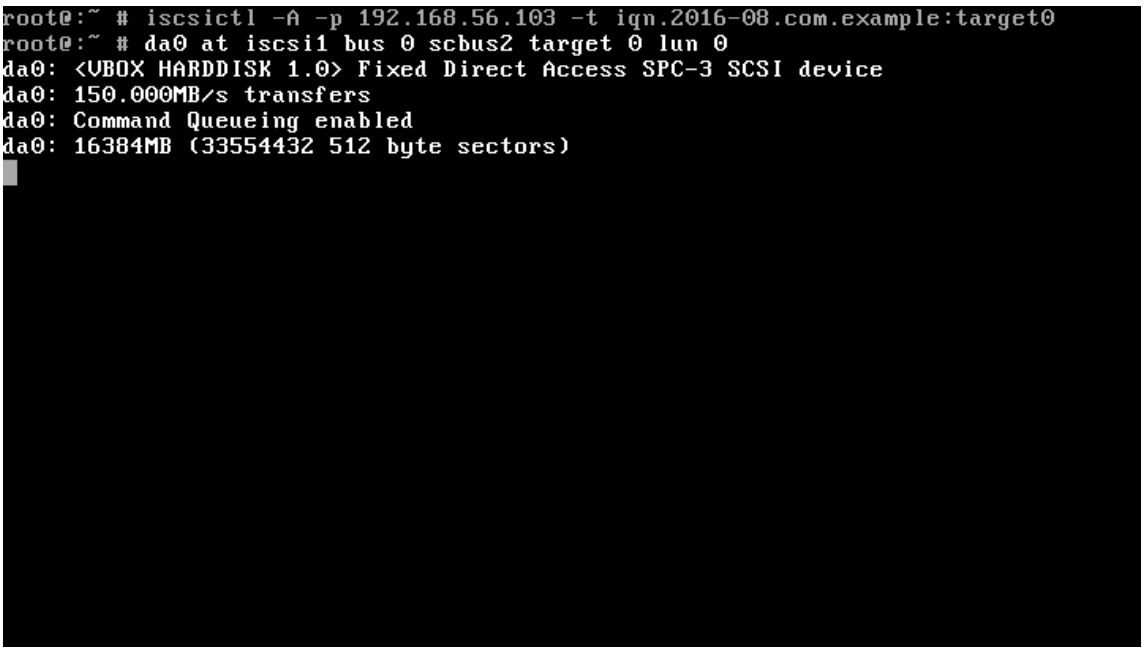

Figure 24. Initiator connecting to Host using iscsictl command

Once it is successful kernel detects new CTL SCSI device and assigns a peripheral device node based on the SCSI device.

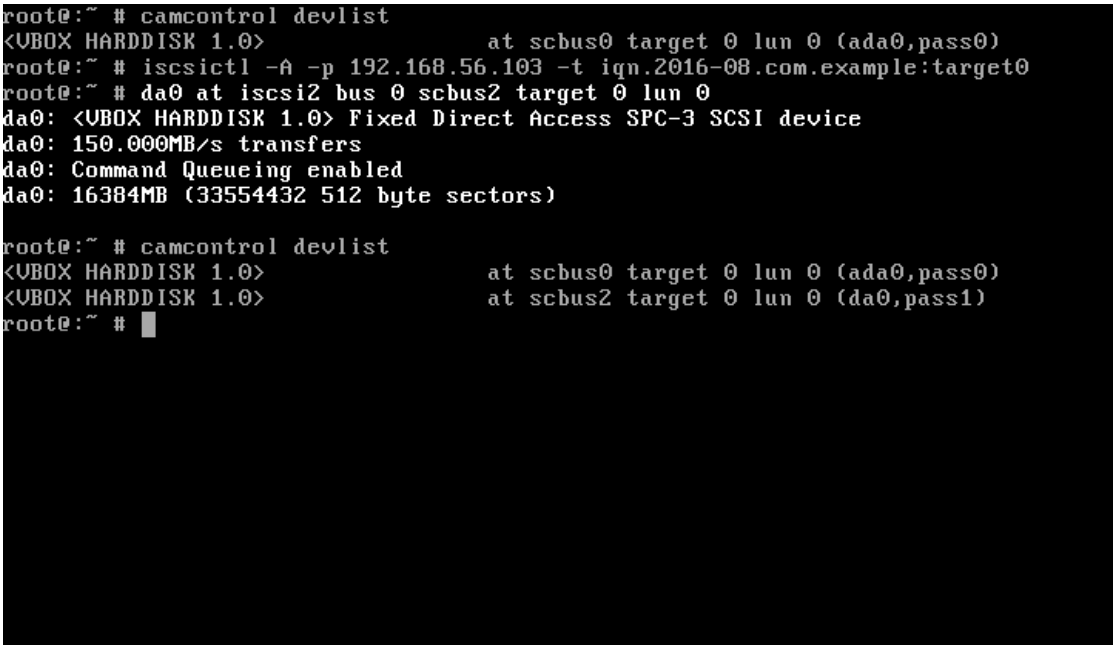

Figure 25. Camcontrol devlist command before and after iscsictl command

Attached screenshot has camcontrol devlist commands before and after iSCSI connection. Camcontrol devlist command display all the devices attached to system.

We can format the device from the initiator side.

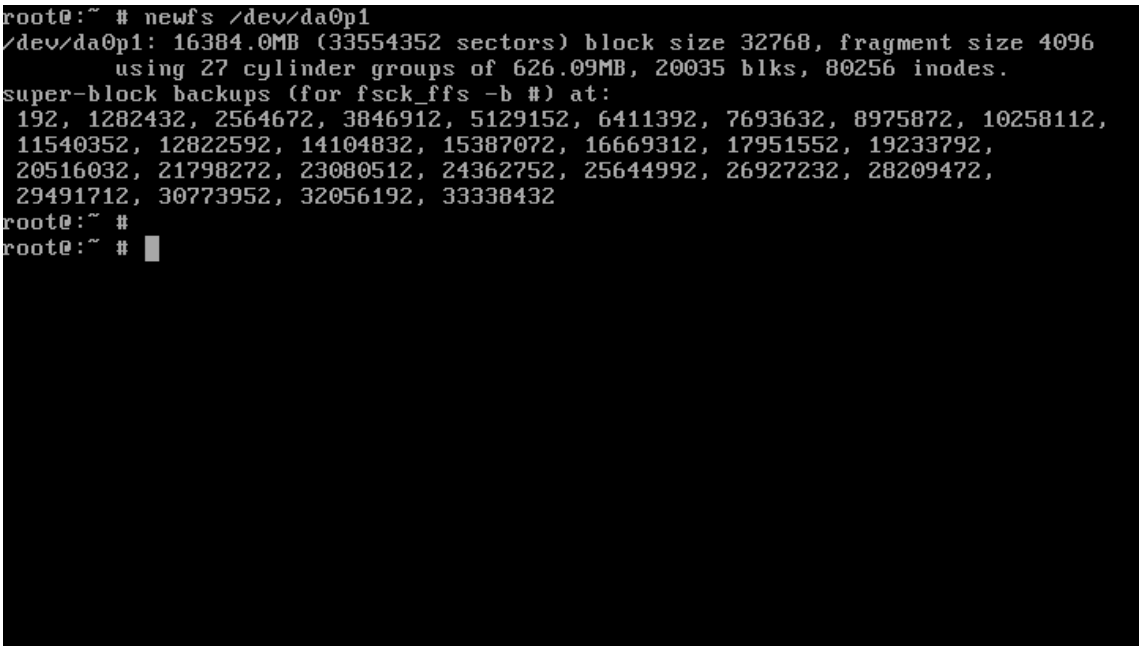

Figure 26. Formatting CTL disk from initiator side

On the initiator side, we can use this as an external drive for storing data.

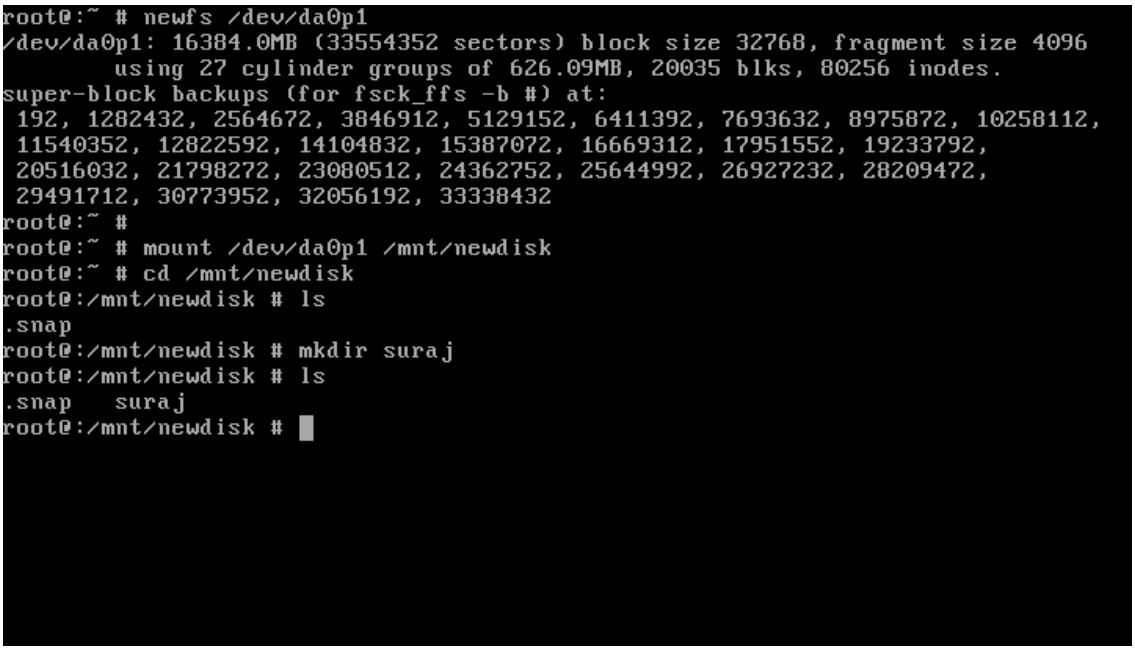

Figure 27. Mounting disk to the initiator file system

On the initiator side, client can use this device as a backup device or by using dd command client can copy data from one folder or drive to CTL drive or vice versa.

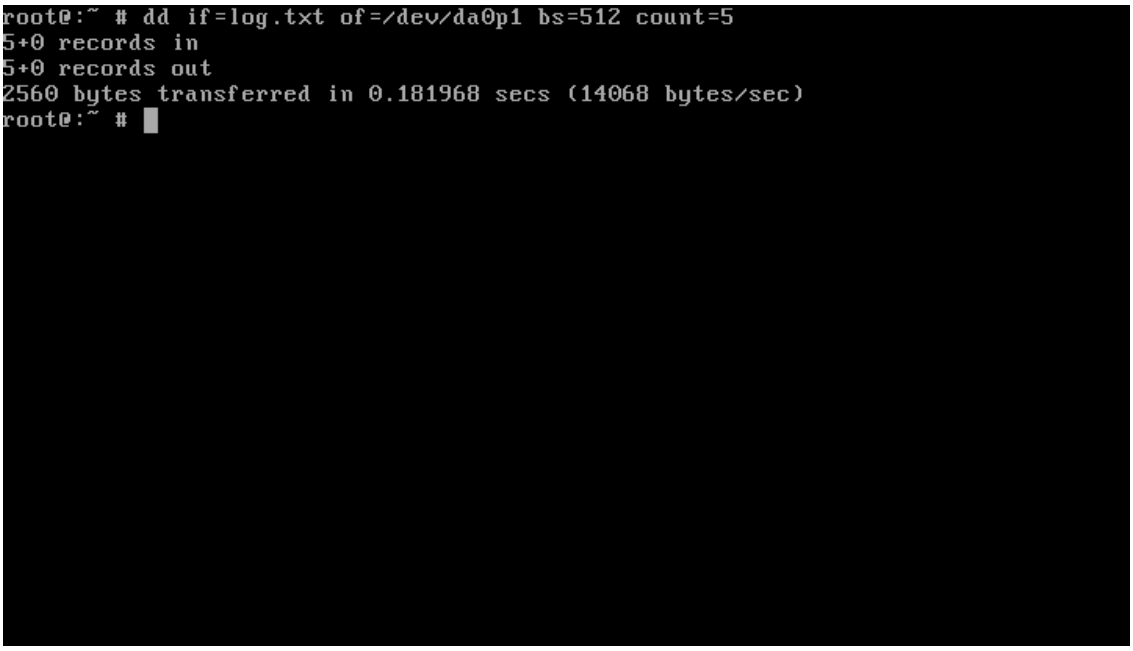

Figure 28. Using dd command to copy data in and out of the disk

By using iscsictl -L command, client can see all the iSCSI connections including host name and IP address of the host device [14].

Figure 29. Using iscsictl command to show the connected devices through iSCSI

Chapter 5

CONCLUSION

This section talks about the lessons I learnt in the process of doing this project. During the project, I learnt to write kernel drivers in FreeBSD Operating System and understood internals of kernel. I also learnt debugging in kernel using gdb when the system crashes. I learnt how iSCSI and SCSI devices work.

REFERENCES

[1] FreeBSD. iscsictl manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=iscsictl&sektion=8 Accessed in April 2016.

[2] FreeBSD. iscsid manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=iscsid&sektion=8&apropos=0&manpath=FreeBSD+10.3-RELEASE+and+Ports Accessed in April 2016.

[3] FreeBSD. ctl.conf manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=ctl.conf&sektion=5 Accessed in April 2016.

[4] FreeBSD. ctld manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=ctld&sektion=8&apropos=0&manpath=FreeBSD+10.3-RELEASE+and+Ports Accessed in April 2016.

[5] FreeBSD. ctladm manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=ctladm&sektion=8&apropos=0&manpath=FreeBSD+10.3-RELEASE+and+Ports Accessed in April 2016.

[6] FreeBSD. camcontrol manual. [Online]. Available: https://www.freebsd.org/cgi/man.cgi?query=camcontrol&sektion=8&apropos=0&manpath=FreeBSD+10.3-RELEASE+and+Ports Accessed in May 2016.

[7] FreeBSD. Design of FreeBSD SCSI subsystem. [Online]. Available: https://people.freebsd.org/~gibbs/ARTICLE-0001.html Accessed in May 2016.

[8] Wikipedia. SCSI command. [Online]. Available: https://en.wikipedia.org/wiki/SCSI_command Accessed in June 2016.

[9] Wikipedia. SCSI. [Online]. Available: https://en.wikipedia.org/wiki/SCSI Accessed in June 2016.

[10] Wikipedia. iSCSI. [Online]. Available: https://en.wikipedia.org/wiki/ISCSI Accessed in June 2016.

[11] GitHub. CTL. [Online]. Available: https://github.com/suraj5/freebsd/blob/master/sys/cam/ctl/README.ctl.txt Accessed in July 2016.

[12] Dtrace. About dtrace. [Online]. Available: http://dtrace.org/blogs/about/ Accessed in July 2016.

[13] Gnu. GDB debugger. [Online]. Available: https://www.gnu.org/software/gdb/ Accessed in July 2016.

[14] FreeBSD. About FreeBSD. [Online] Available: https://www.freebsd.org/about.html Accessed in July 2016