实验二 SPDK安装和使用

**一、实验目的**

1. 学习SPDK基本原理和用法
2. 学习SPDK ZNS SSD相关API使用方法

**二、实验内容**

1. 下载SPDK源代码并编译安装
2. 运行NVME hello world程序
3. 通过分析NVME hello world源码学习SPDK基本原理
4. 修改hello world，实现zns命令I/O读写

**三、实验过程和步骤**

运⾏NVME hello world程序：

sudo scripts/pkgdep.sh –all

./configure

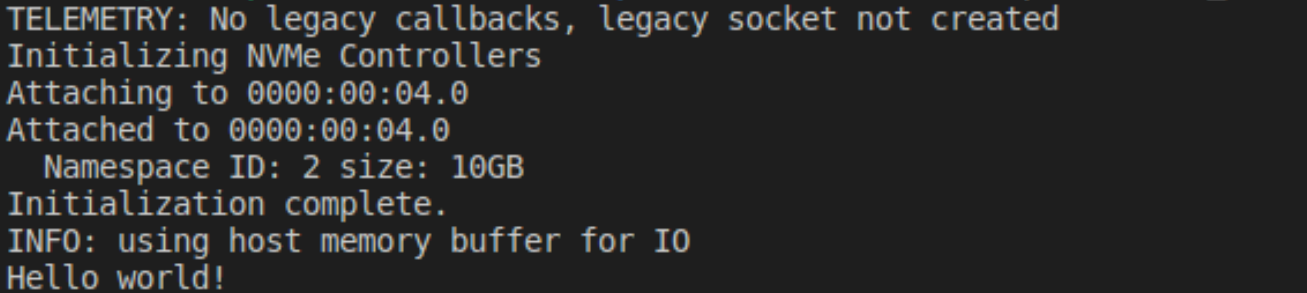
Make

/test/unit/unittest.sh

sudo make install

sudo scripts/setup.sh

sudo ./build/examples/hello\_world



分析NVME hello world源码：

函数处理流程： main()：

int main(int argc, char \*\*argv)

{

int rc;

struct spdk\_env\_opts opts;

\*

\* SPDK relies on an abstraction around the local environment

\* named env that handles memory allocation and PCI device operations.

\* This library must be initialized first.

\*

/

spdk\_env\_opts\_init(&opts);

rc = parse\_args(argc, argv, &opts);

if (rc = 0) {

return rc;

}

opts.name = "hello\_world";

if (spdk\_env\_init(&opts) < 0) {

fprintf(stderr, "Unable to initialize SPDK env\n");

return 1;

}

printf("Initializing NVMe Controllers\n");

if (g\_vmd & spdk\_vmd\_init()) {

fprintf(stderr, "Failed to initialize VMD."

" Some NVMe devices can be unavailable.\n");

}

\*

\* Start the SPDK NVMe enumeration process. probe\_cb will be called

\* for each NVMe controller found, giving our application a choice on

\* whether to attach to each controller. attach\_cb will then be

\* called for each controller after the SPDK NVMe driver has completed

\* initializing the controller we chose to attach.

/

rc = spdk\_nvme\_probe(&g\_trid, NULL, probe\_cb, attach\_cb, NULL);

if (rc = 0) {

fprintf(stderr, "spdk\_nvme\_probe() failed\n");

rc = 1;

goto exit;

}

if (TAILQ\_EMPTY(&g\_controllers)) {

fprintf(stderr, "no NVMe controllers found\n");

rc = 1;

goto exit;

}

printf("Initialization complete.\n");

hello\_world();

cleanup();

if (g\_vmd) {

spdk\_vmd\_fini();

}

exit:

cleanup();

spdk\_env\_fini();

return rc;

}

Ps:

spdk\_env\_opts\_init() 初始化opts参数

spdk\_env\_init() 初始化环境

spdk\_nvme\_probe() 加载NVMe设备

hello\_world() 进⾏读写操作

cleanup() 释放NVMe设备

hello\_world():

static void hello\_world(void)

{

struct ns\_entry \*ns\_entry;

struct hello\_world\_sequence sequence;

int rc;

size\_t sz;

TAILQ\_FOREACH(ns\_entry, &g\_namespaces, link) {

\*

\* Allocate an I/O qpair that we can use to submit read/write

requests

\* to namespaces on the controller. NVMe controllers typically

support

\* many qpairs per controller. Any I/O qpair allocated for a

controller

\* can submit I/O to any namespace on that controller.

\*

\* The SPDK NVMe driver provides no synchronization for qpair

accesses -

\* the application must ensure only a single thread submits I/O to a

\* qpair, and that same thread must also check for completions on

that

\* qpair. This enables extremely efficient I/O processing by making

all

\* I/O operations completely lockless.

/

ns\_entry > qpair = spdk\_nvme\_ctrlr\_alloc\_io\_qpair(ns\_entry > ctrlr,

NULL, 0);

if (ns\_entry > qpair = NULL) {

printf("ERROR: spdk\_nvme\_ctrlr\_alloc\_io\_qpair() failed\n");

return;

}

\*

\* Use spdk\_dma\_zmalloc to allocate a 4KB zeroed buffer. This memory

\* will be pinned, which is required for data buffers used for SPDK

NVMe

\* I/O operations.

/

sequence.using\_cmb\_io = 1;

sequence.buf = spdk\_nvme\_ctrlr\_map\_cmb(ns\_entry > ctrlr, &sz);

if (sequence.buf = NULL | sz < 0x1000) {

sequence.using\_cmb\_io = 0;

sequence.buf = spdk\_zmalloc(0x1000, 0x1000, NULL,

SPDK\_ENV\_SOCKET\_ID\_ANY, SPDK\_MALLOC\_DMA);

}

if (sequence.buf = NULL) {

printf("ERROR: write buffer allocation failed\n");

return;

}

if (sequence.using\_cmb\_io) {

printf("INFO: using controller memory buffer for IO\n");

} else {

printf("INFO: using host memory buffer for IO\n");

}

sequence.is\_completed = 0;

sequence.ns\_entry = ns\_entry;

\*

\* If the namespace is a Zoned Namespace, rather than a regular

\* NVM namespace, we need to reset the first zone, before we

\* write to it. This not needed for regular NVM namespaces.

/

if (spdk\_nvme\_ns\_get\_csi(ns\_entry > ns) = SPDK\_NVME\_CSI\_ZNS) {

reset\_zone\_and\_wait\_for\_completion(&sequence);

}

\*

\* Print "Hello world!" to sequence.buf. We will write this data to

LBA

\* 0 on the namespace, and then later read it back into a separate

buffer

\* to demonstrate the full I/O path.

/

snprintf(sequence.buf, 0x1000, "%s", "Hello world!\n");

\*

\* Write the data buffer to LBA 0 of this namespace.

"write\_complete" and

\* "&sequence" are specified as the completion callback function and

\* argument respectively. write\_complete() will be called with the

\* value of &sequence as a parameter when the write I/O is

completed.

\* This allows users to potentially specify different completion

\* callback routines for each I/O, as well as pass a unique handle

\* as an argument so the application knows which I/O has completed.

\*

\* Note that the SPDK NVMe driver will only check for completions

\* when the application calls spdk\_nvme\_qpair\_process\_completions().

\* It is the responsibility of the application to trigger the

polling

\* process.

/

rc = spdk\_nvme\_ns\_cmd\_write(ns\_entry > ns, ns\_entry > qpair,

sequence.buf,

0, \* LBA start /

1, \* number of LBAs /

write\_complete, &sequence, 0);

if (rc = 0) {

fprintf(stderr, "starting write I/O failed\n");

exit(1);

}

\*

\* Poll for completions. 0 here means process all available

completions.

\* In certain usage models, the caller may specify a positive

integer

\* instead of 0 to signify the maximum number of completions it

should

\* process. This function will never block - if there are no

\* completions pending on the specified qpair, it will return

immediately.

\*

\* When the write I/O completes, write\_complete() will submit a new

I/O

\* to read LBA 0 into a separate buffer, specifying read\_complete()

as its

\* completion routine. When the read I/O completes, read\_complete()

will

\* print the buffer contents and set sequence.is\_completed = 1.

That will

\* break this loop and then exit the program.

/

while (!sequence.is\_completed) {

spdk\_nvme\_qpair\_process\_completions(ns\_entry > qpair, 0);

}

\*

\* Free the I/O qpair. This typically is done when an application

exits.

\* But SPDK does support freeing and then reallocating qpairs during

\* operation. It is the responsibility of the caller to ensure all

\* pending I/O are completed before trying to free the qpair.

/

spdk\_nvme\_ctrlr\_free\_io\_qpair(ns\_entry > qpair);

}

}

Ps：

spdk\_nvme\_ctrlr\_alloc\_io\_qpair() 为控制器分配I/O qpair

spdk\_zmalloc() 分配buffer

snprintf(sequence.buf, . ) 写⼊数据到buffer

spdk\_nvme\_ns\_cmd\_write() 从buffer写⼊namespace的LBA 0处

spdk\_nvme\_qpair\_process\_completions() 处理I/O Completions

spdk\_nvme\_ctrlr\_free\_io\_qpair 释放I/O qpair

write\_complete()：

static void write\_complete(void \*arg, const struct spdk\_nvme\_cpl \*completion)

{

struct hello\_world\_sequence \*sequence = arg;

struct ns\_entry \*ns\_entry = sequence > ns\_entry;

int rc;

spdk\_free()

spdk\_zmalloc()

spdk\_nvme\_ns\_cmd\_read()

\*

See if an error occurred. If so, display information

\* about it, and set completion value so that I/O

\* caller is aware that an error occurred.

/

if (spdk\_nvme\_cpl\_is\_error(completion)) {

spdk\_nvme\_qpair\_print\_completion(sequence > ns\_entry > qpair, (struct

spdk\_nvme\_cpl \*)completion);

fprintf(stderr, "I/O error status: %s\n",

spdk\_nvme\_cpl\_get\_status\_string(&completion > status));

fprintf(stderr, "Write I/O failed, aborting run\n");

sequence > is\_completed = 2;

exit(1);

}

\*

\* The write I/O has completed. Free the buffer associated with

\* the write I/O and allocate a new zeroed buffer for reading

\* the data back from the NVMe namespace.

/

if (sequence > using\_cmb\_io) {

spdk\_nvme\_ctrlr\_unmap\_cmb(ns\_entry > ctrlr);

} else {

spdk\_free(sequence > buf);

}

sequence > buf = spdk\_zmalloc(0x1000, 0x1000, NULL,

SPDK\_ENV\_SOCKET\_ID\_ANY, SPDK\_MALLOC\_DMA);

rc = spdk\_nvme\_ns\_cmd\_read(ns\_entry > ns, ns\_entry > qpair, sequence > buf,

0, \* LBA start /

1, \* number of LBAs /

read\_complete, (void \*)sequence, 0);

if (rc = 0) {

fprintf(stderr, "starting read I/O failed\n");

exit(1);

}

}

Ps:

spdk\_free() 释放write时分配的buffer

\*

See if an error occurred. If so, display information

\* about it, and set completion value so that I/O

\* caller is aware that an error occurred.

/

if (spdk\_nvme\_cpl\_is\_error(completion)) {

spdk\_nvme\_qpair\_print\_completion(sequence > ns\_entry > qpair, (struct

spdk\_nvme\_cpl \*)completion);

fprintf(stderr, "I/O error status: %s\n",

spdk\_nvme\_cpl\_get\_status\_string(&completion > status));

fprintf(stderr, "Write I/O failed, aborting run\n");

sequence > is\_completed = 2;

exit(1);

}

\*

\* The write I/O has completed. Free the buffer associated with

\* the write I/O and allocate a new zeroed buffer for reading

\* the data back from the NVMe namespace.

/

if (sequence > using\_cmb\_io) {

spdk\_nvme\_ctrlr\_unmap\_cmb(ns\_entry > ctrlr);

} else {

spdk\_free(sequence > buf);

}

sequence > buf = spdk\_zmalloc(0x1000, 0x1000, NULL,

SPDK\_ENV\_SOCKET\_ID\_ANY, SPDK\_MALLOC\_DMA);

rc = spdk\_nvme\_ns\_cmd\_read(ns\_entry > ns, ns\_entry > qpair, sequence > buf,

0, \* LBA start /

1, \* number of LBAs /

read\_complete, (void \*)sequence, 0);

if (rc = 0) {

fprintf(stderr, "starting read I/O failed\n");

exit(1);

}

}

sequence->is\_completed = 1

printf(sequence->buf)

spdk\_free()

spdk\_zmalloc() 为read分配buffer

spdk\_nvme\_ns\_cmd\_read() 将namespace的LBA 0读⼊到buffer中

read\_complete():

static void read\_complete(void \*arg, const struct spdk\_nvme\_cpl \*completion)

{

struct hello\_world\_sequence \*sequence = arg;

\*

Assume the I/O was successful /

sequence > is\_completed = 1;

\*

See if an error occurred. If so, display information

\* about it, and set completion value so that I/O

\* caller is aware that an error occurred.

/

if (spdk\_nvme\_cpl\_is\_error(completion)) {

spdk\_nvme\_qpair\_print\_completion(sequence > ns\_entry > qpair, (struct

spdk\_nvme\_cpl \*)completion);

fprintf(stderr, "I/O error status: %s\n",

spdk\_nvme\_cpl\_get\_status\_string(&completion > status));

fprintf(stderr, "Read I/O failed, aborting run\n");

sequence > is\_completed = 2;

exit(1);

}

\*

\* The read I/O has completed. Print the contents of the

\* buffer, free the buffer, then mark the sequence as

\* completed. This will trigger the hello\_world() function

\* to exit its polling loop.

/

printf("%s", sequence > buf);

spdk\_free(sequence > buf);

}

Ps：

sequence->is\_completed = 1 修改完成标识，使hello\_world中while退出

printf("%s", sequence->buf) 将buffer中数据输出

spdk\_free() 释放read的buffer

修改hello world，实现zns命令I/O读写：

hello\_miracle.c:

#include "spdk/stdinc.h"

#include "spdk/nvme.h"

#include "spdk/vmd.h"

#include "spdk/nvme\_zns.h"

#include "spdk/env.h"

#include "spdk/string.h"

#include "spdk/log.h"

struct ctrlr\_entry

{

struct spdk\_nvme\_ctrlr \*ctrlr;

TAILQ\_ENTRY(ctrlr\_entry) link;

char name[1024];

};

struct ns\_entry

{

struct spdk\_nvme\_ctrlr \*ctrlr;

struct spdk\_nvme\_ns \*ns;

TAILQ\_ENTRY(ns\_entry) link;

struct spdk\_nvme\_qpair \*qpair;

};

struct my\_sequence

{

struct ns\_entry \*ns\_entry;

char \*buf;

unsigned using\_cmb\_io;

int is\_completed;

};

static TAILQ\_HEAD(, ctrlr\_entry) g\_controllers =

TAILQ\_HEAD\_INITIALIZER(g\_controllers);

static TAILQ\_HEAD(, ns\_entry) g\_namespaces =

TAILQ\_HEAD\_INITIALIZER(g\_namespaces);

static struct spdk\_nvme\_transport\_id g\_trid = {};

static bool g\_vmd = false;

static void register\_ns(struct spdk\_nvme\_ctrlr \*ctrlr, struct spdk\_nvme\_ns

\*ns)

{

struct ns\_entry \*entry;

if (!spdk\_nvme\_ns\_is\_active(ns))

{

return;

}

entry = malloc(sizeof(struct ns\_entry));

if (entry = NULL)

{

perror("ns\_entry malloc");

exit(1);

}

entry > ctrlr = ctrlr;

entry > ns = ns;

TAILQ\_INSERT\_TAIL(&g\_namespaces, entry, link);

printf(" Namespace ID: %d size: %juGB\n", spdk\_nvme\_ns\_get\_id(ns),

spdk\_nvme\_ns\_get\_size(ns) / 1000000000);

}

static bool probe\_cb(void \*cb\_ctx, const struct spdk\_nvme\_transport\_id \*trid,

struct spdk\_nvme\_ctrlr\_opts \*opts)

{

printf("Attaching to %s\n", trid > traddr);

return true;

}

static void attach\_cb(void \*cb\_ctx, const struct spdk\_nvme\_transport\_id

\*trid, struct spdk\_nvme\_ctrlr \*ctrlr, const struct spdk\_nvme\_ctrlr\_opts

\*opts)

{

int nsid;

struct ctrlr\_entry \*entry;

struct spdk\_nvme\_ns \*ns;

const struct spdk\_nvme\_ctrlr\_data \*cdata;

entry = malloc(sizeof(struct ctrlr\_entry));

if (entry = NULL)

{

perror("ctrlr\_entry malloc");

exit(1);

}

printf("Attached to %s\n", trid > traddr);

cdata = spdk\_nvme\_ctrlr\_get\_data(ctrlr);

snprintf(entry > name, sizeof(entry > name), "%-20.20s (%-20.20s)", cdata-

>mn, cdata > sn);

entry > ctrlr = ctrlr;

TAILQ\_INSERT\_TAIL(&g\_controllers, entry, link);

for (nsid = spdk\_nvme\_ctrlr\_get\_first\_active\_ns(ctrlr); nsid = 0;

nsid = spdk\_nvme\_ctrlr\_get\_next\_active\_ns(ctrlr, nsid))

{

ns = spdk\_nvme\_ctrlr\_get\_ns(ctrlr, nsid);

if (ns = NULL)

{

continue;

}

register\_ns(ctrlr, ns);

}

}

static void cleanup(void)

{

struct ns\_entry \*ns\_entry, \*tmp\_ns\_entry;

struct ctrlr\_entry \*ctrlr\_entry, \*tmp\_ctrlr\_entry;

struct spdk\_nvme\_detach\_ctx \*detach\_ctx = NULL;

TAILQ\_FOREACH\_SAFE(ns\_entry, &g\_namespaces, link, tmp\_ns\_entry)

{

TAILQ\_REMOVE(&g\_namespaces, ns\_entry, link);

free(ns\_entry);

}

TAILQ\_FOREACH\_SAFE(ctrlr\_entry, &g\_controllers, link, tmp\_ctrlr\_entry)

{

TAILQ\_REMOVE(&g\_controllers, ctrlr\_entry, link);

spdk\_nvme\_detach\_async(ctrlr\_entry > ctrlr, &detach\_ctx);

free(ctrlr\_entry);

}

if (detach\_ctx)

{

spdk\_nvme\_detach\_poll(detach\_ctx);

}

}

static void usage(const char \*program\_name)

{

printf("%s [options]", program\_name);

printf("\t\n");

printf("options:\n");

printf("\t[-d DPDK huge memory size in MB]\n");

printf("\t[-g use single file descriptor for DPDK memory segments]\n");

printf("\t[-i shared memory group ID]\n");

printf("\t[-r remote NVMe over Fabrics target address]\n");

printf("\t[-V enumerate VMD]\n");

#ifdef DEBUG

printf("\t[-L enable debug logging]\n");

#else

printf("\t[-L enable debug logging (flag disabled, must reconfigure with

enable-debug)]\n");

#endif

}

static int parse\_args(int argc, char \*\*argv, struct spdk\_env\_opts \*env\_opts)

{

int op, rc;

spdk\_nvme\_trid\_populate\_transport(&g\_trid, SPDK\_NVME\_TRANSPORT\_PCIE);

snprintf(g\_trid.subnqn, sizeof(g\_trid.subnqn), "%s",

SPDK\_NVMF\_DISCOVERY\_NQN);

while ((op = getopt(argc, argv, "d:gi:r:L:V")) = -1)

{

switch (op)

{

case 'V':

g\_vmd = true;

break;

case 'i':

env\_opts > shm\_id = spdk\_strtol(optarg, 10);

if (env\_opts > shm\_id < 0)

{

fprintf(stderr, "Invalid shared memory ID\n");

return env\_opts > shm\_id;

}

break;

case 'g':

env\_opts > hugepage\_single\_segments = true;

break;

case 'r':

if (spdk\_nvme\_transport\_id\_parse(&g\_trid, optarg) = 0)

{

fprintf(stderr, "Error parsing transport address\n");

return 1;

}

break;

case 'd':

env\_opts > mem\_size = spdk\_strtol(optarg, 10);

if (env\_opts > mem\_size < 0)

{

fprintf(stderr, "Invalid DPDK memory size\n");

return env\_opts > mem\_size;

}

break;

case 'L':

rc = spdk\_log\_set\_flag(optarg);

if (rc < 0)

{

fprintf(stderr, "unknown flag\n");

usage(argv[0]);

exit(EXIT\_FAILURE);

}

#ifdef DEBUG

spdk\_log\_set\_print\_level(SPDK\_LOG\_DEBUG);

#endif

break;

default:

usage(argv[0]);

return 1;

}

}

return 0;

}

static void reset\_zone\_complete(void \*arg, const struct spdk\_nvme\_cpl

\*completion)

{

struct my\_sequence \*sequence = arg;

sequence > is\_completed = 1;

if (spdk\_nvme\_cpl\_is\_error(completion)) {

spdk\_nvme\_qpair\_print\_completion(sequence > ns\_entry > qpair, (struct

spdk\_nvme\_cpl \*)completion);

fprintf(stderr, "I/O error status: %s\n",

spdk\_nvme\_cpl\_get\_status\_string(&completion > status));

fprintf(stderr, "Reset zone I/O failed, aborting run\n");

sequence > is\_completed = 2;

exit(1);

}

}

static void reset\_zone\_and\_wait\_for\_completion(struct my\_sequence \*sequence)

{

if (spdk\_nvme\_zns\_reset\_zone(sequence > ns\_entry > ns, sequence > ns\_entry-

>qpair,

0, \* starting LBA of the zone to reset /

false, \* don't reset all zones /

reset\_zone\_complete,

sequence)) {

fprintf(stderr, "starting reset zone I/O failed\n");

exit(1);

}

while (!sequence > is\_completed) {

spdk\_nvme\_qpair\_process\_completions(sequence > ns\_entry > qpair, 0);

}

sequence > is\_completed = 0;

}

bool io\_completed;

static void check\_completion(void \*arg, const struct spdk\_nvme\_cpl \*cpl)

{

if (spdk\_nvme\_cpl\_is\_error(cpl))

{

printf("I/O Option Failed\n");

}

io\_completed = true;

}

static void print\_zns\_zone(uint8\_t \*report, uint32\_t index, uint32\_t zdes)

{

struct spdk\_nvme\_zns\_zone\_desc \*desc;

uint32\_t i, zds, zrs, zd\_index;

zrs = sizeof(struct spdk\_nvme\_zns\_zone\_report);

zds = sizeof(struct spdk\_nvme\_zns\_zone\_desc);

zd\_index = zrs + index \* (zds + zdes);

desc = (struct spdk\_nvme\_zns\_zone\_desc \*)(report + zd\_index);

printf("ZSLBA: 0x%016"PRIx64" ZCAP: 0x%016"PRIx64" WP: 0x%016"PRIx64" ZS:

", desc > zslba,

desc > zcap, desc > wp);

switch (desc > zs) {

case SPDK\_NVME\_ZONE\_STATE\_EMPTY:

printf("Empty");

break;

case SPDK\_NVME\_ZONE\_STATE\_IOPEN:

printf("Implicit open");

break;

case SPDK\_NVME\_ZONE\_STATE\_EOPEN:

printf("Explicit open");

break;

case SPDK\_NVME\_ZONE\_STATE\_CLOSED:

printf("Closed");

break;

case SPDK\_NVME\_ZONE\_STATE\_RONLY:

printf("Read only");

break;

case SPDK\_NVME\_ZONE\_STATE\_FULL:

printf("Full");

break;

case SPDK\_NVME\_ZONE\_STATE\_OFFLINE:

printf("Offline");

break;

default:

printf("Reserved");

}

printf(" ZT: %s ZA: %x\n", (desc > zt = SPDK\_NVME\_ZONE\_TYPE\_SEQWR) ?

"SWR" : "Reserved",

desc > za.raw);

if (!desc > za.bits.zdev) {

return;

}

for (i = 0; i < zdes; i += 8) {

printf("zone\_desc\_ext[%d] : 0x%"PRIx64"\n", i,

\*(uint64\_t \*)(report + zd\_index + zds + i));

}

}

static void hello\_miracle(void)

{

struct ns\_entry \*ns\_entry;

struct my\_sequence sequence;

int rc;

size\_t sz;

ns\_entry = g\_namespaces.tqh\_first;

ns\_entry > qpair = spdk\_nvme\_ctrlr\_alloc\_io\_qpair(ns\_entry > ctrlr, NULL,

0);

if (ns\_entry > qpair = NULL) {

printf("ERROR: spdk\_nvme\_ctrlr\_alloc\_io\_qpair() failed\n");

return;

}

sequence.using\_cmb\_io = 1;

sequence.buf = spdk\_nvme\_ctrlr\_map\_cmb(ns\_entry > ctrlr, &sz);

if (sequence.buf = NULL | sz < 0x1000) {

sequence.using\_cmb\_io = 0;

sequence.buf = spdk\_zmalloc(0x1000, 0x1000, NULL,

SPDK\_ENV\_SOCKET\_ID\_ANY, SPDK\_MALLOC\_DMA);

}

if (sequence.buf = NULL) {

printf("ERROR: write buffer allocation failed\n");

return;

}

if (sequence.using\_cmb\_io) {

printf("INFO: using controller memory buffer for IO\n");

} else {

printf("INFO: using host memory buffer for IO\n");

}

sequence.is\_completed = 0;

sequence.ns\_entry = ns\_entry;

reset\_zone\_and\_wait\_for\_completion(&sequence);

\*

\* @brief zns ssd info

\*

/

uint64\_t num\_zones = spdk\_nvme\_zns\_ns\_get\_num\_zones(ns\_entry > ns);

uint64\_t zone\_size = spdk\_nvme\_zns\_ns\_get\_zone\_size(ns\_entry > ns);

uint32\_t zone\_append\_size\_limit =

spdk\_nvme\_zns\_ctrlr\_get\_max\_zone\_append\_size(ns\_entry > ctrlr);

const struct spdk\_nvme\_ns\_data \*ref\_ns\_data =

spdk\_nvme\_ns\_get\_data(ns\_entry > ns);

const struct spdk\_nvme\_zns\_ns\_data \*ref\_ns\_zns\_data =

spdk\_nvme\_zns\_ns\_get\_data(ns\_entry > ns);

printf("\*\*\*\*\*\*\*\*\*\*\*\* NVMe Information \*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Number of Zone: %lu\n", num\_zones);

printf("Size of LBA: %lu\n", ref\_ns\_data > nsze);

printf("Size of Zone: %lu (%lu \* %lu)\n", zone\_size, ref\_ns\_zns\_data-

>lbafe > zsze, ref\_ns\_data > nsze);

printf("Append Size Limit of Zone: %u\n", zone\_append\_size\_limit);

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

uint8\_t \*report\_buf;

size\_t report\_buf\_size;

uint64\_t nr\_zones = 0;

uint64\_t max\_zones\_per\_buf;

uint32\_t zds, zrs, zd\_index;

size\_t zdes = 0;

zrs = sizeof(struct spdk\_nvme\_zns\_zone\_report);

zds = sizeof(struct spdk\_nvme\_zns\_zone\_desc);

report\_buf\_size = spdk\_nvme\_ns\_get\_max\_io\_xfer\_size(ns\_entry > ns);

report\_buf = calloc(1, report\_buf\_size);

if (!report\_buf)

{

printf("Zone report allocation failed!\n");

return;

}

memset(report\_buf, 0, report\_buf\_size);

max\_zones\_per\_buf = (report\_buf\_size - zrs) / zds;

rc = spdk\_nvme\_zns\_report\_zones(ns\_entry > ns, ns\_entry > qpair,

report\_buf, report\_buf\_size, 0, SPDK\_NVME\_ZRA\_LIST\_ALL, true,

check\_completion, NULL);

if (rc)

{

fprintf(stderr, "Report zones failed\n");

return;

}

io\_completed = false;

while (!io\_completed)

{

spdk\_nvme\_qpair\_process\_completions(ns\_entry > qpair, 0);

}

nr\_zones = report\_buf[0];

if (nr\_zones > max\_zones\_per\_buf)

{

fprintf(stderr, "nr\_zones too big\n");

return;

}

if (!nr\_zones)

{

return;

}

printf("\*\*\*\*\*\*\*\*\*\*\*\* Zone Information \*\*\*\*\*\*\*\*\*\*\*\*\n");

uint32\_t i;

for (i = 0; i < nr\_zones & i < num\_zones; + i)

{

print\_zns\_zone(report\_buf, i, zdes);

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

struct spdk\_nvme\_zns\_zone\_desc \*first\_zone\_info;

zd\_index = zrs + 0 \* (zds + zdes);

first\_zone\_info = (struct spdk\_nvme\_zns\_zone\_desc \*)(report\_buf +

zd\_index);

printf("Writing Data to Buffer . \n");

snprintf(sequence.buf, 0x1000, "%s", "Hello Miracle!\n");

printf("Writing Buffer to the first LBA of the first Zone . \n");

io\_completed = false;

rc = spdk\_nvme\_zns\_zone\_append(ns\_entry > ns, ns\_entry > qpair,

sequence.buf, first\_zone\_info > zslba, 1, check\_completion, NULL, 0);

if (rc = 0) {

fprintf(stderr, "starting write I/O failed\n");

exit(1);

}

while (!io\_completed) {

spdk\_nvme\_qpair\_process\_completions(ns\_entry > qpair, 0);

}

printf("Finish Writing!\n");

printf("Reading Data from the first LBA of the first Zone . \n");

spdk\_free(sequence.buf);

sequence.buf = spdk\_zmalloc(0x1000, 0x1000, NULL, SPDK\_ENV\_SOCKET\_ID\_ANY,

SPDK\_MALLOC\_DMA);

io\_completed = false;

rc = spdk\_nvme\_ns\_cmd\_read(ns\_entry > ns, ns\_entry > qpair, sequence.buf,

first\_zone\_info > zslba, 1, check\_completion, NULL, 0);

if (rc = 0) {

fprintf(stderr, "starting read I/O failed\n");

exit(1);

}

while (!io\_completed) {

spdk\_nvme\_qpair\_process\_completions(ns\_entry > qpair, 0);

}

printf("Finish Reading, Data is: %s", sequence.buf);

spdk\_free(sequence.buf);

free(report\_buf);

spdk\_nvme\_ctrlr\_free\_io\_qpair(ns\_entry > qpair);

}

int main(int argc, char \*\*argv)

{

int rc;

struct spdk\_env\_opts opts;

spdk\_env\_opts\_init(&opts);

rc = parse\_args(argc, argv, &opts);

if (rc = 0)

{

return rc;

}

opts.name = "hello\_miracle";

if (spdk\_env\_init(&opts) < 0)

{

fprintf(stderr, "Unable to initialize SPDK env\n");

return 1;

}

printf("Initializing NVMe Controllers\n");

if (g\_vmd & spdk\_vmd\_init())

{

fprintf(stderr, "Failed to initialize VMD."

" Some NVMe devices can be unavailable.\n");

}

rc = spdk\_nvme\_probe(&g\_trid, NULL, probe\_cb, attach\_cb, NULL);

if (rc = 0)

{

fprintf(stderr, "spdk\_nvme\_probe() failed\n");

rc = 1;

goto exit;

}

if (TAILQ\_EMPTY(&g\_controllers))

{

fprintf(stderr, "no NVMe controllers found\n");

rc = 1;

goto exit;

}

printf("Initialization complete.\n");

hello\_miracle();

/

test();

cleanup();

if (g\_vmd)

{

spdk\_vmd\_fini();

}

exit:

cleanup();

spdk\_env\_fini();

return rc;

}

Makefile:

SPDK\_ROOT\_DIR := $(abspath $(CURDIR)/ . /spdk)

APP = hello\_miracle

include $(SPDK\_ROOT\_DIR)/mk/nvme.libtest.mk

run: all

@ rm -f hello\_miracle.d hello\_miracle.o

@ sudo ./hello\_miracle

运行：make run

**四、实验结论和心得体会**

本次实验下载SPDK源代码并编译安装，运行NVME hello world程序，通过分析NVME hello world源码学习了SPDK基本原理，并修改hello world，实现了zns命令I/O读写。