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

3.1 SPDK 编译

Follow 如下命令：

git clone https://github.com/spdk/spdk

cd spdk

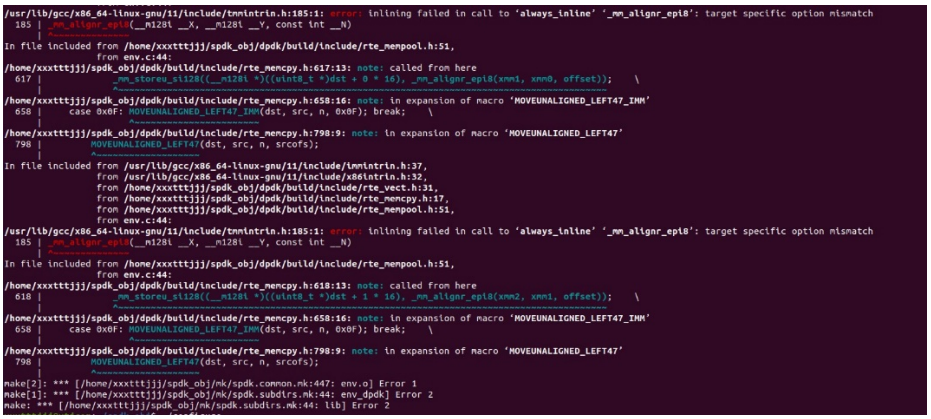
git submodule update –init

sudo scripts/pkgdep.sh –all

./configure

make

如果遇到 make 报错，需重新启动 Ubuntu 并添加命令 -cpu qemu64,+ssse3,+sse4.1,+sse4.2 (reference: qq 群内同学答疑)



3.2 运行 NVME hello world 程序

完成页分配和设备解绑：

sudo make install

sudo scripts/setup.sh

sudo HUGEMEM=4096 scripts/setup.sh

启用 helloworld：

```
zhp@zhp:~/exp/spdk$ sudo scripts/setup.sh
0000:00:04.0 (1b36 0010): nvme -> uio_pci_generic
zhp@zhp:~/exp/spdk$ sudo HUGEMEM=4096 scripts/setup.sh
0000:00:04.0 (1b36 0010): Already using the uio_pci_generic driver
zhp@zhp:~/exp/spdk$
zhp@zhp:~/exp/spdk$ sudo ./build/examples/hello_world
TELEMETRY: No legacy callbacks, legacy socket not created
Initializing NVMe Controllers
Attaching to 0000:00:04.0
Attached to 0000:00:04.0
  Namespace ID: 2 size: 10GB
Initialization complete.
INFO: using host memory buffer for IO
Hello world!
zhp@zhp:~/exp/spdk$
```

3.3 分析 hello world

Hello world 的 main 函数源码如下

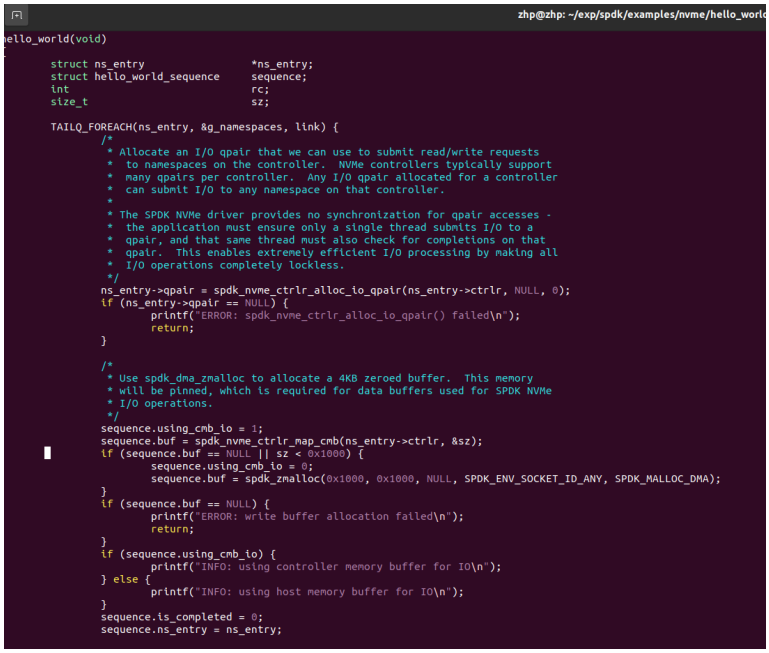
zhp@zhp: ~/exp/spdk/examples/nvme/hello_world

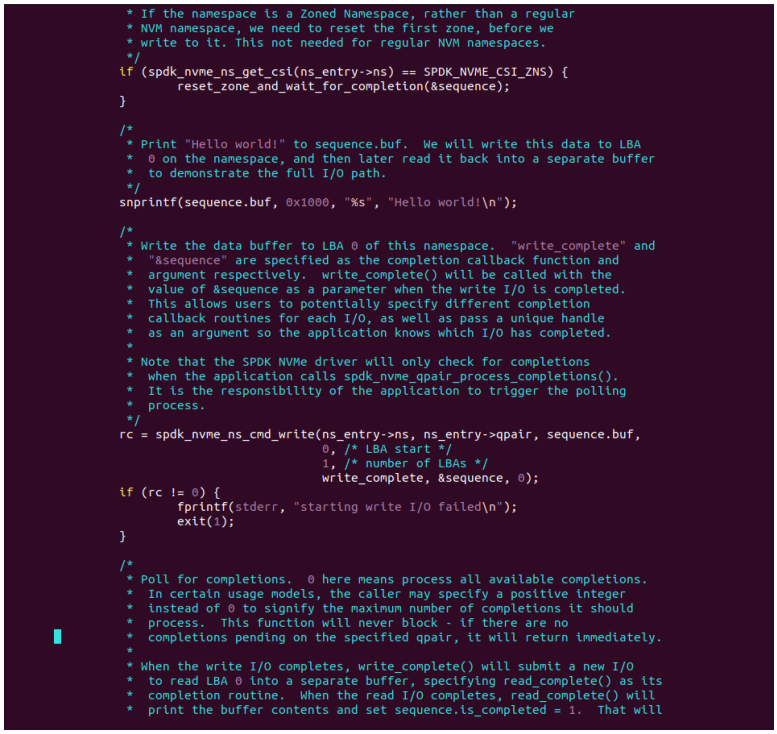
```
main(int argc, char **argv)
{
        int rc;
        struct spdk_env_opts opts;

        /*
         * 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;
        }
```

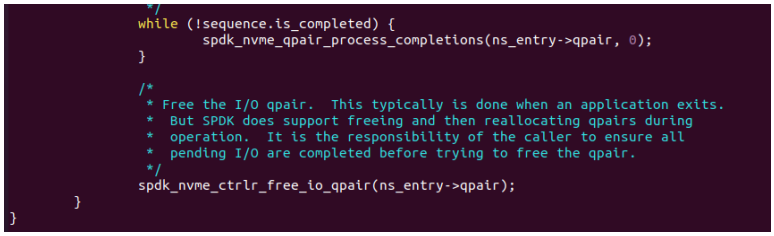
```
        printf("Initialization complete.\n");
        hello_world();
        cleanup();
        if (g_vmd) {
                spdk_vmd_fini();
        }

exit:
        cleanup();
        spdk_env_fini();
        return rc;
}
```

spdk_env_opts_init()和spdk_env_init()进行环境的初始化设置，而spdk_nvme_probe() 加载NVMe设备，hello_world()进行读写操作，cleanup()释放NVME设备。

查看hello world细节







spdk_nvme_ctrlr_alloc_io_qpair() 为控制器分配IO qpairs，spdk_zmalloc() 分配空间,snprintf(sequence.buf, ...) 写入数据，spdk_nvme_ns_cmd_write()从buffer写入namespace的LAB 0处，spdk_nvme_qpair_process_completions()处理IO 是否完成进程。

当IO完成时，write_complete()将调用spdk_nvme_ns_cmd_read()进行读取，读取后，read_complete()将打印"hello world“即缓存区的内容。

3.4 修改 hello world，实现 zns 命令 I/O 读写

目录文件一览

```
zhp@zhp:~/exp/task2$ ls
Makefile  zhp  zhp.c  zhp.d  zhp.o
zhp@zhp:~/exp/task2$ cd ..
zhp@zhp:~/exp$ ls
output  spdk  task2
zhp@zhp:~/exp$
```

仿照 hello world 文件，在 task2 中编写 zhp.c 并读取打印“hello zhp_test!”,代码如下：

```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_zhp(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 zhp_test!\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_zhp";

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_zhp();

// test();


cleanup();
```

```
    if (g_vmd)
    {
        spdk_vmd_fini();
    }

exit:
    cleanup();
    spdk_env_fini();
    return rc;
}
```

Makefile 文件:

```
zhp@zhp: ~/exp/task2
SPDK_ROOT_DIR := $(abspath $(CURDIR)/../spdk)

APP =zhp

include $(SPDK_ROOT_DIR)/mk/nvme.libtest.mk

run: all
	@ sudo ./zhp
```

得到部分结果如下：

```
zhp@zhp:~/exp/task2$ vim Makefile
zhp@zhp:~/exp/task2$ make run
/bin/sh: 1: pkg-config: not found
  CC task2/zhp.o
  LINK zhp
TELEMETRY: No legacy callbacks, legacy socket not created
Initializing NVMe Controllers
Attaching to 0000:00:04.0
Attached to 0000:00:04.0
  Namespace ID: 2 size: 10GB
Initialization complete.
INFO: using host memory buffer for IO
************ NVMe Information ************
Number of Zone: 81920
Size of LBA: 2621440
Size of Zone: 131072 (32 * 2621440)
Append Size Limit of Zone: 524288
****************** END ******************
************ Zone Information ************
```

```
ZSLBA: 0x0000000000001ea0 ZCAP: 0x0000000000000020 WP: 0x0000000000001ea0 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001ec0 ZCAP: 0x0000000000000020 WP: 0x0000000000001ec0 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001ee0 ZCAP: 0x0000000000000020 WP: 0x0000000000001ee0 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001f00 ZCAP: 0x0000000000000020 WP: 0x0000000000001f00 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001f20 ZCAP: 0x0000000000000020 WP: 0x0000000000001f20 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001f40 ZCAP: 0x0000000000000020 WP: 0x0000000000001f40 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001f60 ZCAP: 0x0000000000000020 WP: 0x0000000000001f60 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001f80 ZCAP: 0x0000000000000020 WP: 0x0000000000001f80 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001fa0 ZCAP: 0x0000000000000020 WP: 0x0000000000001fa0 ZS: Empty ZT: SWR ZA: 0
ZSLBA: 0x0000000000001fc0 ZCAP: 0x0000000000000020 WP: 0x0000000000001fc0 ZS: Empty ZT: SWR ZA: 0
****************** END ******************
Writing Data to Buffer ...
Writing Buffer to the first LBA of the first Zone ...
Finish Writing!
Reading Data from the first LBA of the first Zone ...
Finish Reading, Data is: Hello zhp_test!
zhp@zhp:~/exp/task2$
```

四、实验结论和心得体会

通过本次实验，学习了 SPDK 基本原理和用法以及 ZNS SSD 相关 API 使用方法，实现了 zns 命令 I/O 读写。