实验六 综合设计实验

一、实验目的

1. 基于SPDK，完成一个综合性设计实验，理解底层NVMe设备驱动到应用程序之间I/O栈。
2. 掌握文件系统或KV数据库原理并进行实现。

二、实验内容

1. 设计一个兼容POSIX语义的简易文件系统，可在测试程序中完成fopen, fclose, fread, fwrite, fseek。
2. 在BlobFS中实现文件系统内置压缩、加密功能。
3. 基于BlobFS，设计一个key-value数据库，支持open，put，get，close操作。
4. 如果只想及格，可在SPDK搭建成功前提下，分析BlobFS或FTL源码，给出代码分析报告。可重点选择一个函数，从顶向下分析其I/O栈，例如，对于BlobFS的spdk_file_write（）函数，层层分析其调用栈，直到底层nvme驱动调用，画出函数调用图。

三、实验代码及结果

时间原因，本次实验只分析源码部分

`spdk_file_write()` 是BlobFS中的一个函数，用于将数据写入文件。经过不断的向下调用函数，会负责将数据缓存到内存中，然后使用NVMe驱动将数据写入存储设备。

官方文档的参数描述如下，包括文件句柄、数据缓冲区、偏移量和数据长度。

spdk_file_write()

```
int spdk_file_write (
        struct spdk_file *          file,
        struct spdk_fs_thread_ctx * ctx,
        void *                      payload,
        uint64_t                    offset,
        uint64_t                    length
)
```

Write data to the given file.

Parameters

file	File to write.
ctx	The thread context for this operation
payload	The specified buffer which should contain the data to be transmitted.
offset	The beginning position to write data.
length	The size in bytes of data to write.

Returns

`spdk_file_write()` 源码如下

```
int
```

```
spdk_file_write(struct spdk_file *file, struct spdk_fs_thread_ctx *ctx,
        void *payload, uint64_t offset, uint64_t length)
{
    struct spdk_fs_channel *channel = (struct spdk_fs_channel *)ctx;
    struct spdk_fs_request *flush_req;
    uint64_t rem_length, copy, blob_size, cluster_sz;
    uint32_t cache_buffers_filled = 0;
    uint8_t *cur_payload;
    struct cache_buffer *last;

    BLOBFS_TRACE_RW(file, "offset=%jx length=%jx\n", offset, length);

    if (length == 0) {
        return 0;
    }

    if (offset != file->append_pos) {
        BLOBFS_TRACE(file, " error offset=%jx append_pos=%jx\n", offset, file-
>append_pos);
        return -EINVAL;
    }

    pthread_spin_lock(&file->lock);
    file->open_for_writing = true;

    if ((file->last == NULL) && (file->append_pos % CACHE_BUFFER_SIZE == 0)) {
        cache_append_buffer(file);
    }

    if (file->last == NULL) {
        struct rw_from_file_arg arg = {};
        int rc;

        arg.channel = channel;
        arg.rwerrno = 0;
        file->append_pos += length;
        pthread_spin_unlock(&file->lock);
        rc = __send_rw_from_file(file, payload, offset, length, false, &arg);
        if (rc != 0) {
            return rc;
        }
        sem_wait(&channel->sem);
        return arg.rwerrno;
    }

    blob_size = __file_get_blob_size(file);

    if ((offset + length) > blob_size) {
        struct spdk_fs_cb_args extend_args = {};

        cluster_sz = file->fs->bs_opts.cluster_sz;
        extend_args.sem = &channel->sem;
        extend_args.op.resize.num_clusters = __bytes_to_clusters((offset +
length), cluster_sz);
        extend_args.file = file;
```

```
        BLOBFS_TRACE(file, "start resize to %u clusters\n",
extend_args.op.resize.num_clusters);
        pthread_spin_unlock(&file->lock);
        file->fs->send_request(__file_extend_blob, &extend_args);
        sem_wait(&channel->sem);
        if (extend_args.rc) {
            return extend_args.rc;
        }
    }

    flush_req = alloc_fs_request(channel);
    if (flush_req == NULL) {
        pthread_spin_unlock(&file->lock);
        return -ENOMEM;
    }

    last = file->last;
    rem_length = length;
    cur_payload = payload;
    while (rem_length > 0) {
        copy = last->buf_size - last->bytes_filled;
        if (copy > rem_length) {
            copy = rem_length;
        }
        BLOBFS_TRACE_RW(file, "  fill offset=%jx length=%jx\n", file-
>append_pos, copy);
        memcpy(&last->buf[last->bytes_filled], cur_payload, copy);
        file->append_pos += copy;
        if (file->length < file->append_pos) {
            file->length = file->append_pos;
        }
        cur_payload += copy;
        last->bytes_filled += copy;
        rem_length -= copy;
        if (last->bytes_filled == last->buf_size) {
            cache_buffers_filled++;
            last = cache_append_buffer(file);
            if (last == NULL) {
                BLOBFS_TRACE(file, "nomem\n");
                free_fs_request(flush_req);
                pthread_spin_unlock(&file->lock);
                return -ENOMEM;
            }
        }
    }

    pthread_spin_unlock(&file->lock);

    if (cache_buffers_filled == 0) {
        free_fs_request(flush_req);
        return 0;
    }

    flush_req->args.file = file;
    file->fs->send_request(__file_flush, flush_req);
```

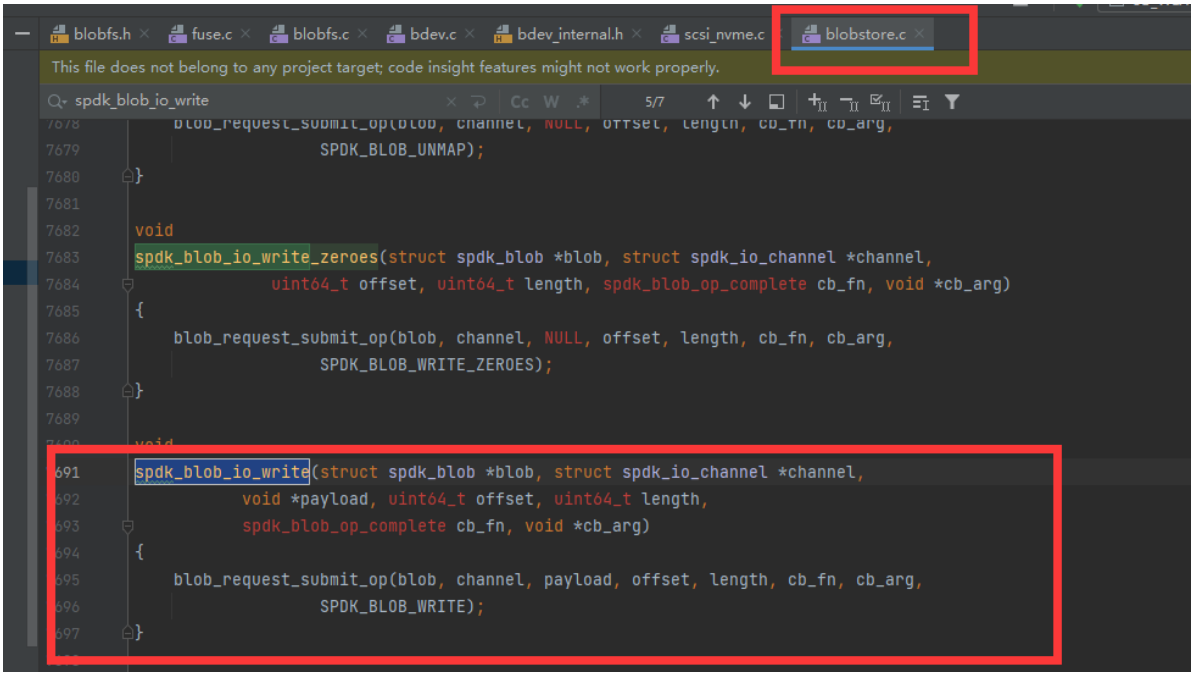
```
    return 0;
}
```

根据代码中的workflow，一层一层向下寻找调用的函数：

当调用`spdk_file_write`时，它会使用`__file_flush`参数调用`send_request`，以异步方式将数据刷新到后端存储。

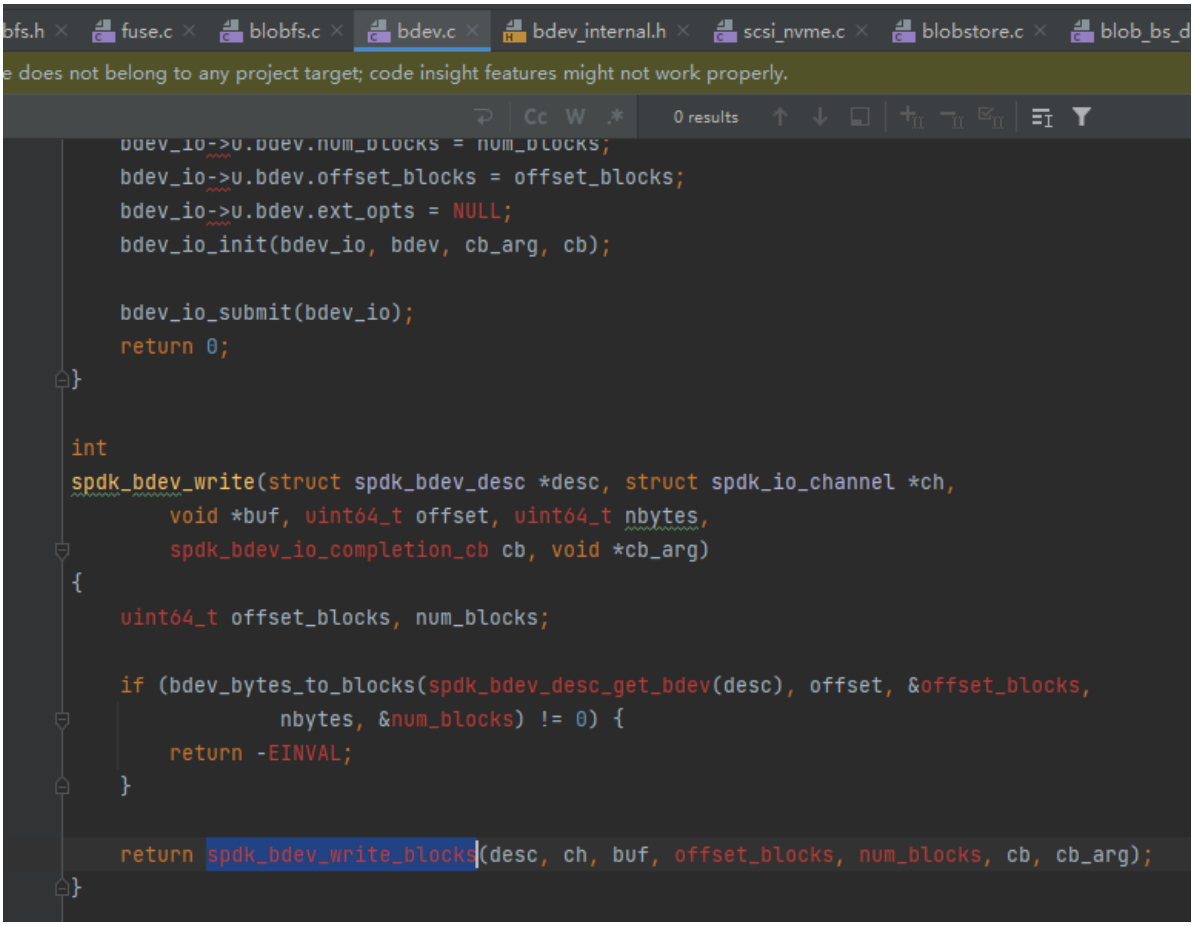
`send_request`将调用`blobstore.c`中的`spdk_blob_io_write`来处理写入请求。

`spdk_blob_io_write`会调用`blob_request_submit_op`，该函数会根据写入请求的大小决定是调用`blob_request_submit_op_single`还是`blob_request_submit_op_split`。

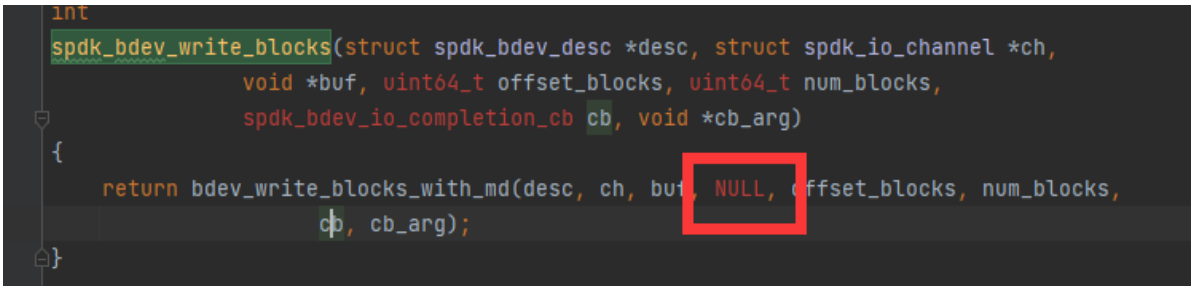


`blob_request_submit_op_single`和`blob_request_submit_op_split`将调用`request.c`中的`bs_sequence_write_dev`或`bs_batch_write_dev`，以将写入请求提交给`blob store`的设备。

`blob store`的设备的写函数`b->bs_dev.write`被设置为`bdev.c`中的`spdk_bdev_write`。此函数将调用`spdk_bdev_write_blocks`将数据写入底层块设备。



`spdk_bdev_write_blocks` 调用 `bdev_write_blocks_with_md`，并将 NULL 作为元数据缓冲区的指针传递给它。



而 `bdev_write_blocks_with_md` 会检查是否允许对块设备进行写操作，并检查写入的块数是否有效。然后，它会创建一个 `bdev_io` 结构体，并将相关信息填充到该结构体中。最后，它会调用 `bdev_io_submit` 函数来提交写操作。

```
bdev_write_blocks_with_md(struct spdk_bdev_desc *desc, struct spdk_io_channel *ch,
              void *buf, void *md_buf, uint64_t offset_blocks, uint64_t num_blocks,
              spdk_bdev_io_completion_cb cb, void *cb_arg)
{
    struct spdk_bdev *bdev = spdk_bdev_desc_get_bdev(desc);
    struct spdk_bdev_io *bdev_io;
    struct spdk_bdev_channel *channel = __io_ch_to_bdev_ch(ch);

    if (!desc->write) {
        return -EBADF;
    }

    if (!bdev_io_valid_blocks(bdev, offset_blocks, num_blocks)) {
        return -EINVAL;
    }

    bdev_io = bdev_channel_get_io(channel);
    if (!bdev_io) {
        return -ENOMEM;
    }

    bdev_io->internal.ch = channel;
    bdev_io->internal.desc = desc;
    bdev_io->type = SPDK_BDEV_IO_TYPE_WRITE;
    bdev_io->u.bdev.iovs = &bdev_io->iov;
    bdev_io->u.bdev.iovs[0].iov_base = buf;
    bdev_io->u.bdev.iovs[0].iov_len = num_blocks * bdev->blocklen;
    bdev_io->u.bdev.iovcnt = 1;
    bdev_io->u.bdev.md_buf = md_buf;
    bdev_io->u.bdev.num_blocks = num_blocks;
    bdev_io->u.bdev.offset_blocks = offset_blocks;
    bdev_io->u.bdev.ext_opts = NULL;
    bdev_io_init(bdev_io, bdev, cb_arg, cb);

    bdev_io_submit(bdev_io);
    return 0;
}
```

`bdev_io_submit` 是一个函数，它用于将一个 bdev_io 结构体提交给底层块设备(`NVME`)进行处理。

其函数调用图从上至下大概如下所示：

```
spdk_file_write
  spdk_file->fs->send_request
    spdk_blob_io_write
      blob_request_submit_op
        blob_request_submit_op_single
          bs_batch_write_dev
            blob_bdev->bs_dev.write
            spdk_bdev_write
              spdk_bdev_write_blocks
                bdev_write_blocks_with_md
```

四、调试和心得体会

本次实验了解了`BlobFS`的`spdk_file_write`函数的调用过程，层层分析了其调用栈，直到底层nvme驱动调用，更加熟悉了SPDK的架构。