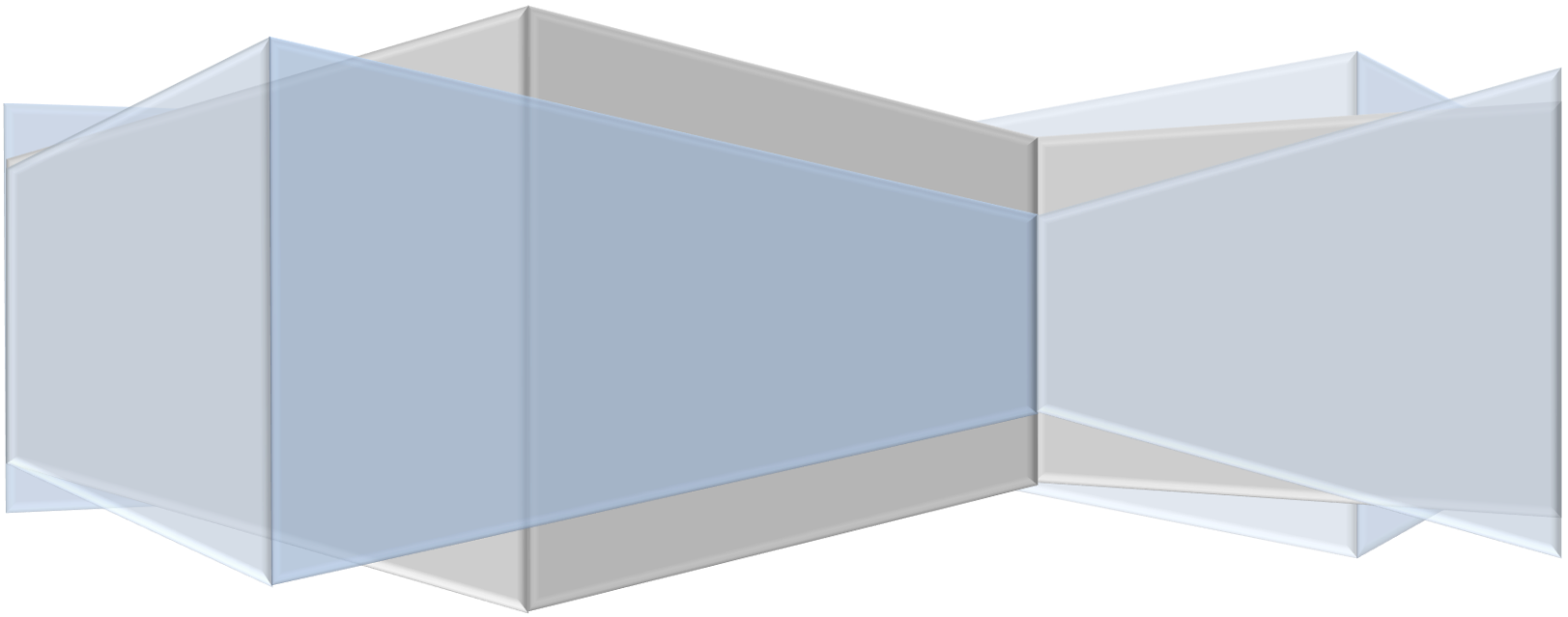
HCL Technologies

KERNEL LEVEL IO TOOL

Block level IO generation



Contents

Introduction

We live in period of time in which I/O performance is extremely important to guaranty stability of computer systems and to allow them to serve millions of users throughout the world or within large organizations.

Many filesystem tools exist to generate IO load and measure IO performance of devices.

This document will focus on the design of a new Kernel level IO tool that generates IO load at block level through low level device driver.

The tool will help to accurately measure the amount of iops for hard drives and SSDs, as IO is generated in the generic block layer.

Existing IO tools

Bonnie, Bonnie++, IoMeter, IoZone, FIO, LADDIS, Netbench, LMBench, VdBench, PostMark, SPEC SFS, IOGen, IOStone, IOBench, Pablo IO Benchmark, NHT-1 I/O Benchmarks, NTIOGen, IOTest, TIOTest, spew, dbench.

Why a new kernel level IO tool is needed

In order to understand the problems in existing IO tools and why a new kernel level IO tool is needed, it is important to know about the linux IO stack.

Linux IO STACK

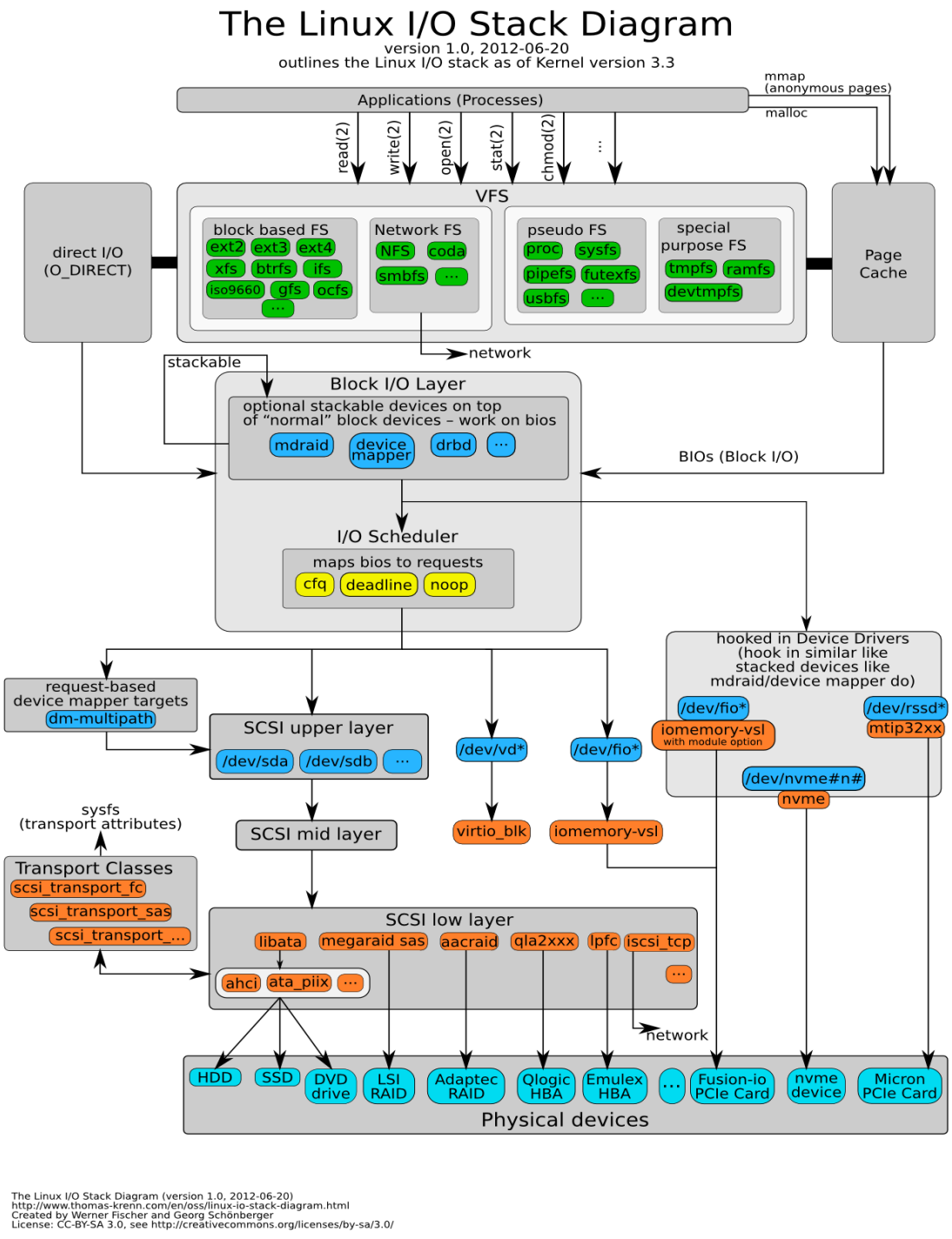


Fig 1. LINUX IO STACK

Application layer: This layer includes applications that run on runtime environments wriiten in C or C++ (tcl,java etc.)

VFS layer: VFS is the highest and most abstract part of kernel's file handling infrastructure. The purpose of this layer is to provide a unified, file-like interface between application and the lower layers.

Filesystem layer: This layer converts high-level operations understood by VFS (reading, writing) into low-level operations on disk blocks

Generic block device layer: This layer is an abstraction for block devices (physical or logical) in the system. It is responsible for receiving IO requests from upper layer and submitting them onto block devices. Features like IO prioritizing (through IO scheduler), remapping of IO requests(for RAID),etc. are provided in this layer.

Device driver: Device driver is the lowest level, least abstract piece of software which interacts with the hardware directly.

Hardware: At the bottom of the stack, real hardware exist.

Problems in existing IO tools

- Exisiting IO tools generate IO load from the top most application layer.
    - Because, IO load is generated from user space, the throughput measured for a device is strongly influenced by excessive number of system calls and context switches
- Though the benchmark tool IOMeter runs below filesystem layer, its performance fluctuates due to effects of buffer cache.
    - When there is excessive dirty data, buffer cache gets flushed and system becomes busy. During this time, throughput becomes zero.

For the above reasons, the existing tools cannot give true performance parameters of the device.

Why kernel level IO tool is advantageous?

- ✓ This tool generates IO at the generic block layer, bypassing VFS, filesystem layer and buffer cache.
- ✓ Since it runs at a kernel level, it eliminates the overhead caused by system calls and context switches.
- ✓ Bypasses many software layers as possible and gets close to the device giving precise performance measurement

TOOL DESCRIPTION

- This tool basically lets user to configure IO load and generates load at block level from a device driver to the block layer.

- On a specific device, IO load can be specified for
  - user-specified duration of time
  - default time period (10 minutes)

- Data to be filled in the target device for write operations can be
  - dynamically filled by the kernel with random data pattern (default)
  - user specified data pattern

- The IO type or the way in which data is to be read/write to the device can be
  - Sequential read (default)
  - Sequential write
  - Random read
  - Random write

HARDWARE AND SOFTWARE REQUIREMENTS

HARDWARE REQUIREMENTS

Block device (hard drive or SSD) on which the IO workload should be run

SOFTWARE REQUIREMENTS

Host running Linux operating system

COMMAND LINE USAGE OF THE TOOL

LOAD THE KERNEL MODULE

Users should load the kernel driver module “kernel_io.ko” using the command,

insmod kernel_io.ko

USER OPTIONS TO RUN THE IO TOOL

Users should run the binary file named “kiotool”

kiotool <file>

LIST OF FILE PARAMETERS

dev=str	Block device on which the IO should be run
rw=str	Type of IO pattern
	Sequential reads : seqread
	Sequential writes: seqwrite
	Random reads : randread
	Random writes : randwrite

runtime=time Tell kiotool to terminate processing after a specified the number of seconds. If not set, IO will run for 5 minutes.

pattern=str If set, kiotool will set the IO buffers with this pattern

Instead of using a workload file, parameters can be given on the command line.

kiotool –dev <block_device> --runtime <time_in_seconds> --rw <IO_type> --pattern <data>

SAMPLE INPUT AND OUTPUT

SAMPLE FILE FORMAT

workload.txt

[work1]

dev=/dev/sda
rw=seqwrite
runtime=300
pattern=”sample pattern to write on the device”

SAMPLE COMMAND LINE USAGE

./kiotool workload.txt

Alternatively,

./kiotool –dev /dev/sda --rw=seqwrite –runtime=300 –pattern=”sample pattern to write on the device”

SAMPLE OUTPUT WHEN THE TOOL IS RUN

./kiotool –dev /dev/sda –runtime 300 –rw seqwrite –pattern

Workload: on device [/dev/sda] with sequential read IO type

Starting IO generation
Total time : 300 seconds [eta 04m:25s]

FLOW FROM APPLICATION TO DRIVER

- User will run the kiotool from command line
- Kiotool will get the user options, namely, block device, runtime, IO type, IO pattern.etc.
- User options will be checked if they are valid.
    - devicename will be checked if the device is a block device using “stat” system call.
    - rw value will be checked if values are seqread, seqwrite, randread or randwrite
- structure user_data will be filled with the values. Structure is defined as follows,

```
struct user_data  {
        char *devicename;
        unsigned long runtime;
        unsigned long pattern;
        bool pattern_based;
        short int rw;
}
```

Default values runtime – 300 seconds, pattern_based – false (in case of write operation), rw – randread, will be filled in the structure, if not specified by the user.

- This structure will be passed on to the miscellaneous or misc driver using ioctl “IO_LOAD”
- Before ioctl is called, a thread will be created in the user program which will keep displaying the elapsed time till the specified time is expired. This will be created using “pthread_create” system call and the associated function “display_time” will display the elapsed time.

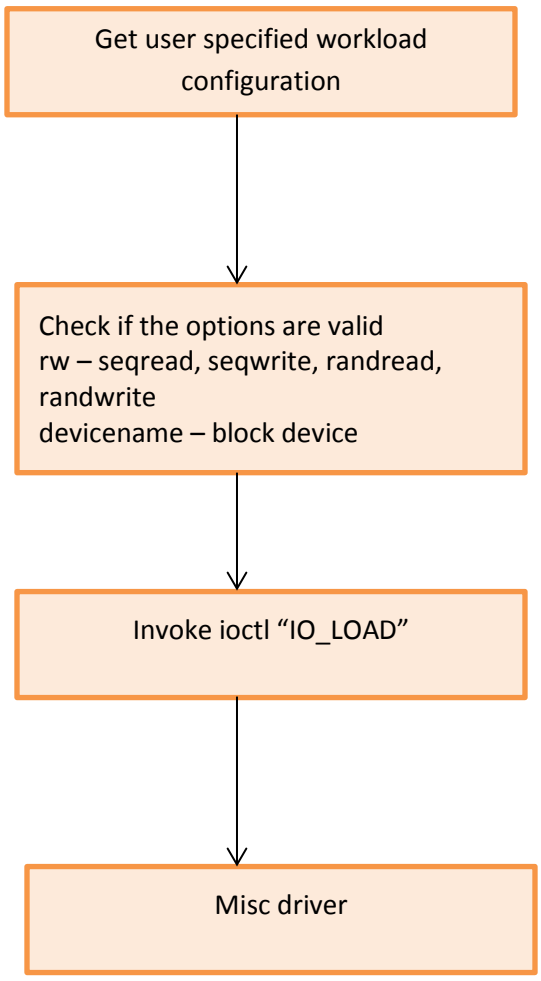


Fig.2. Flow from application to driver

LIST OF KERNEL FUNCTIONS USED IN THIS DESIGN

Workqueue and related API

- Workqueues provide an interface for creating kernel threads that handles and executes queued work. This is used if a schedulable entity is needed to perform processing. They run in process context.

- Workqueue is represented by "struct workqueue_struct".

- Work that is queued to the workqueue is represented by "struct work_struct"

1) **alloc_workqueue** - allocates a new workqueue

    a. int alloc_workqueue(char *name, unsigned int flags, int max_active);

        i. name – name of the queue
        ii. flags - options on how work submitted to the queue will be executed
        iii. max_active - number of tasks that can be executed simultaneously in the queue

2) **destroy_workqueue** – destroys a workqueue

    a. void destroy_workqueue(struct workqueue_struct *wq);

3) **INIT_WORK_ONSTACK** – initializes a work_struct structure

4) **destroy_work_on_stack** – destroys the specified work

    a. void destroy_work_on_stack(struct work_struct *work);

5) **queue_work** – queues work on a workqueue

    a. int queue_work(struct workqueue_struct *queue, struct work_struct *work);

6) **flush_work** - block until a work_struct's callback has terminated

    a. bool flush_work(struct work_struct *work);

Completion and related functions

- Completions are a simple synchronization mechanism that is preferable to sleeping and waking up in some situations. If you have a task that must simply sleep until some process has run its course, completions can do it easily.

- "struct completion" is the structure used for this mechanism.

1) **init_completion** - initializes a completion structure

   void init_completion(struct completion *x);

2) **wait_for_completion** - When driver begins some process whose completion must be waited for, this function is used.

   void wait_for_completion(struct completion *comp);

3) **complete** - When some other part of your code has decided that the completion has happened, it can wake up anybody who is waiting using this function

   void complete(struct completion *comp);

Random number generation and filling in random data:

1) **get_random_bytes** - returns the requested number of random bytes and stores them in a buffer.

   void get_random_bytes(void *buf, int nbytes);

   buf is filled in with random data of size nbytes

2) **randomize_range** – generates a random number in the range [start, end]

unsigned long randomize_range(unsigned long start, unsigned long end, unsigned long len);

The above function is mainly used to generate random address which is page-aligned.

Jiffies

Jiffies is a global variable used in kernel that holds the numberof ticks that have occurred since the system booted. On boot, the kernel initializes the variable to zero, and it is incremented by one during each timer interrupt.Thus, because there are HZ timer interrupts in a second, there are HZ jiffies in a second.The system uptime is therefore jiffies/HZ seconds.

1) **msecs_to_jiffies** – converts time unit milliseconds to jiffies

   unsigned long msecs_to_jiffies(const unsigned int m)

2) **time_before**(unknown, known) macro - returns true if time unknown is before time known; otherwise, it returns false

Atomic operations

1) **atomic_set** – sets an atomic variable

   atomic_set (v, i);

2) **atomic_inc** – increments an atomic variable

   void atomic_inc (atomic_t * v);

3) **atomic_dec_and_test** - Atomically decrements *v* by 1 and returns true if the result is 0, or false for all other cases.

   int atomic_dec_and_test (atomic_t * v);

DESIGN OF THE TOOL IN DRIVER

The driver that handles ioctl request from user space kiotool is a miscellaneous or misc driver. It is a simple character driver having a simplified registration interface. All misc devices are assigned a major number of 10, but can choose a single minor number.

For optimal performance, this driver will

- Be capable of handling IO load on maximums 8 devices concurrently.
- Not run different workload configured from different processes, on the same device
    - That is, if kiotool process is executing sequential writes on device /dev/sda, another kiotool process will not be allowed to execute sequential and random reads or writes on the same device /dev/sda.

INIT MODULE

In kernel_io driver's init module function, a miscellaneous device is registered with the kernel, which is responsible for handling ioctl requests from user space kiotool.

```
int misc_register(struct miscdevice *misc);          /*  interface used for misc registrarion  */
```

CORE DATA STRUCTURES USED IN THE DRIVER:

```
struct user_data {
        char *devicename;               /* name of the device                */
        unsigned long runtime;          /* how long IO will be generated, issued */
        unsigned long pattern;          /* pattern to be filled in the data buffer for
                                           write opeations */
        bool pattern_based;             /* true if pattern is specified by user for
                                           writes */
        short int rw;                   /* IO type – sequential read, sequential
                                           write, random read, random write */
};

struct check_io_end {
        struct completion *io_completion;
        int io_pending;
};
```

"io_pending" parameter is used to indicate whether any IO is pending or not. When this value is non-zero, it means that IO's are still pending.

"io_completion" structure is used to signal that all pending IO's have been completed.

```
struct io_load  {

        struct user_data  udata;

        struct block_device *bdev;          /* representing the target block device */

        unsigned long devsize;              /* size of the target device in bytes */

        struct work_struct *io_work;

        int  running;

        struct check_io_end  end;
};
```

Each work_struct represents the kind of work to be performed on the target device. Typically, the work can be sequential read, sequential write, random read or random write.

"running" parameter is used to indicate whether the work (sequential/random read/write) is currently being executed on the target device.

Initialization of core structures

- "struct io_load" represents IO load configured from the user space kiotool, on a specific device containing runtime, pattern, IO type.etc.
- As the driver will support ioctl requests for maximum 8 devices at a time, memory will be allocated for 8 "struct io_load" structures, using kzalloc function.
- "running" parameter in "struct io_load" will be set to zero.
- A workqueue named "kernel_io_queue" will be created using alloc_workqueue() function, with WQ_HIGHPRI and WQ_CPU_INTENSIVE flags.
    - These flags indicate that any work which is queued on to this workqueue will have high priority and will consume quiet some amount of CPU time.

Handling IOCTL command from user-space

This section will describe how our tool will handle IOCTL requests from user process kiotool.

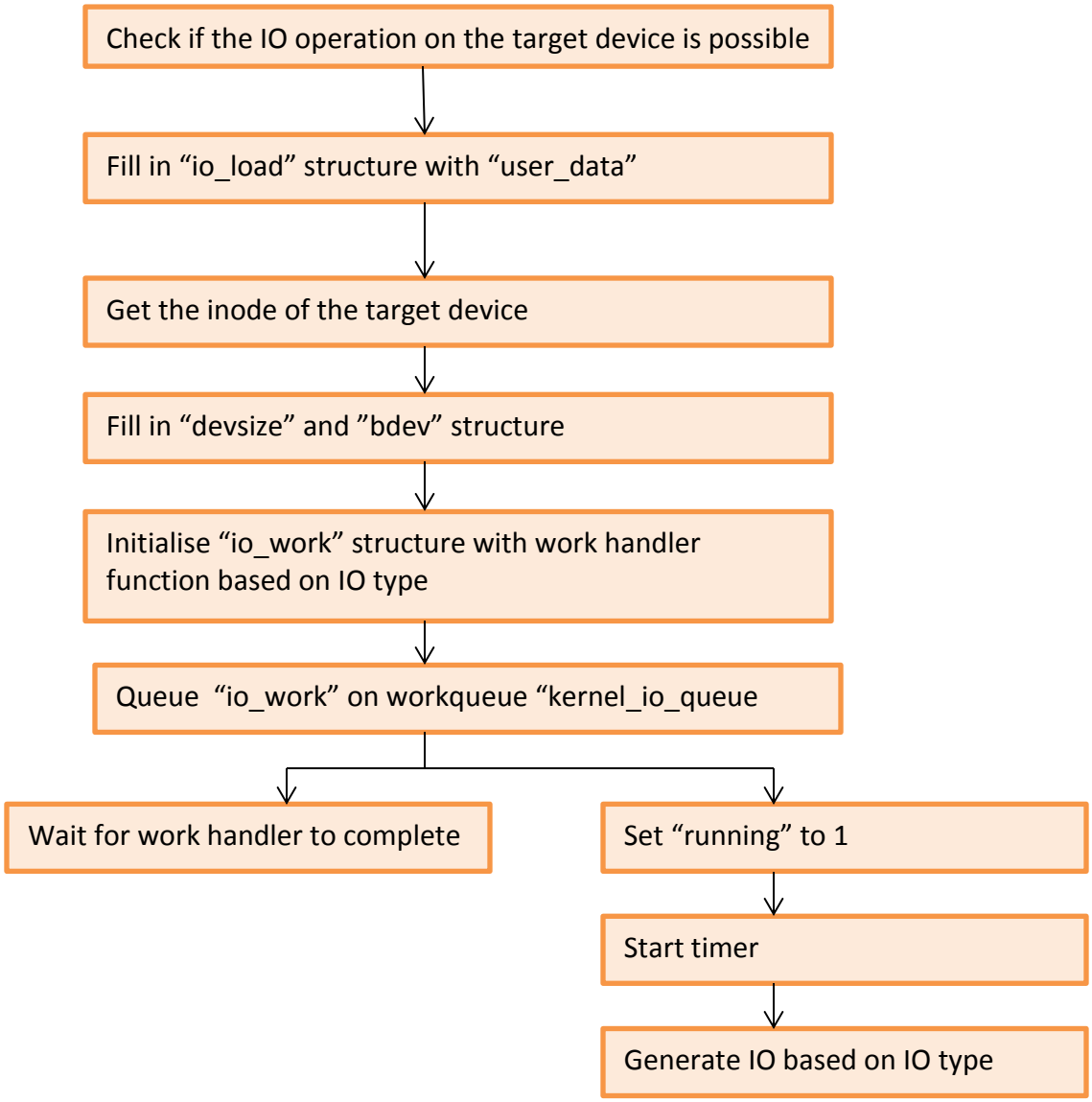


Fig.3. Handling IOCLT command from user-space

int maxCount = 8 /* as driver is capable of handling workload on 8 devices simultaneously*/

1) Check if IO operation is possible:

- Firstly, looping through the maxCount "struct io_load" structures, the driver
    - Checks if driver can handle this request and start IO generation, by checking the value of "running" parameter.
        - If every "struct io_load" structure's "running" parameter is set to 1, then the driver returns error code to application.

    And

    - Checks "devicename" parameter for every structure that has "running" set to 1.
        - If any one of the "io_load" structures indicate that IO is already running on the same device as requested thorugh IOCTL, the driver returns an error code to the application.

2) Fill in "io_load" structure:

- Driver then copies "user_data" structure received from IOCTL to "user_data" structure of a free "io_load" structure, that is not yet used by any other process for IO load.

- Using filp_open() function, driver opens the devicefile.

  struct file *fp = filp_open(devicefile);

- Using "file_inode()" function, driver gets the inode of the target device and its block_device structure.

  inode = file_inode(file);
  struct block_device *bdev = inode->i_bdev;

- devsize- that is device size is filled in using

  i_size_read(inode);

3) Create work based on rw (IO type):

This driver will have 4 types of work, each implementing a work handler function, corresponding to sequential read, sequential write, random read and random write operations.

If the IO type is sequential write,

using INIT_WORK_ONSTACK() function, “io_work” field of “io_load” structure will be initialized to function implementing “sequential write”

4) Queue work

Driver will then queue the high priority, CPU intensive work on the “kernel_io_queue” workqueue created during driver module initialization.

5) Wait for the work to complete:

Initialize “io_completion” completion structure using “init_completion” function.

Wait for the work to be completed in the workqueue handler until the runtime gets elapsed.

This is done using wait_for_completion() function.

Generating BIOs based on IO type in the work handler

In order to understand how sequential and random read or writes are implemented in the driver, it is important to understand the basics of bio structure.

BIO structure:

The basic container for block IO within the kernel is BIO structure. It is the main unit of I/O for the block layer and lower layers (ie drivers and stacking drivers)

struct bio is defined in <linux/bio.h> as,

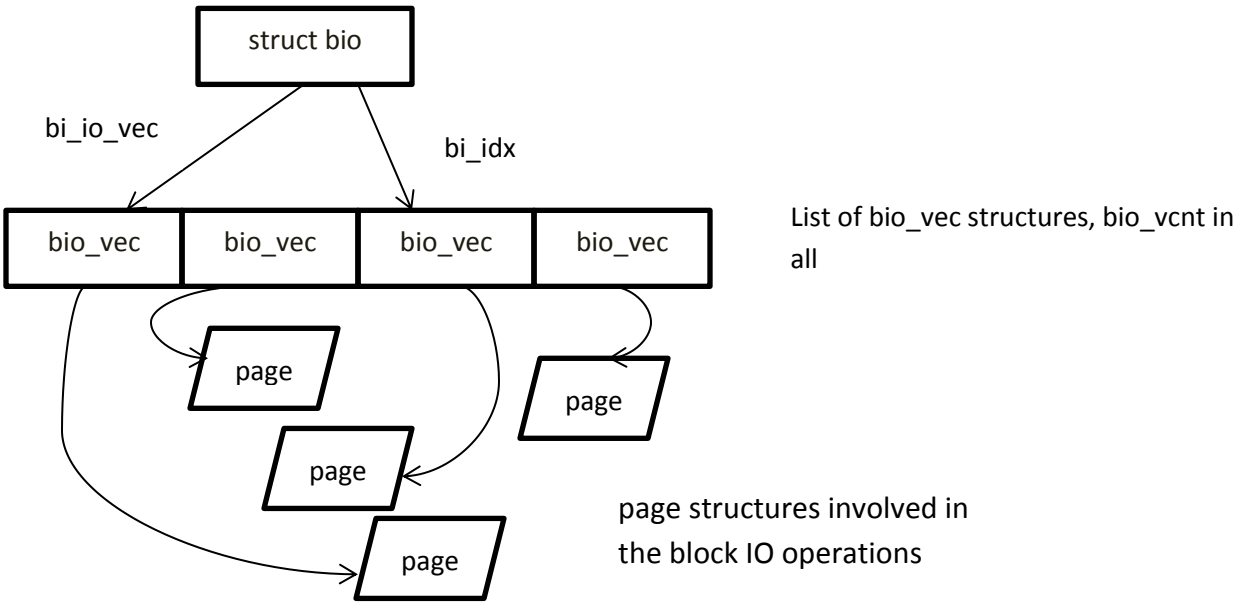
```
struct bio {
        struct bio                  *bi_next;       /* request queue link */
        struct block_device         *bi_bdev;
        unsigned long               bi_flags;       /* status, command, etc */
        unsigned long               bi_rw;          /*  READ/WRITE */
         struct bvec_iter           bi_iter;

        /* Number of segments in this BIO after
        * physical address coalescing is performed.
        */
        unsigned int                bi_phys_segments;

         /*
        * To keep track of the max segment size, we account for the
        * sizes of the first and last mergeable segments in this bio.
         */
        unsigned int                        bi_seg_front_size;
        unsigned int                        bi_seg_back_size;
        atomic_t                    bi_remaining;
        bio_end_io_t                *bi_end_io;
        void                        *bi_private;
        unsigned short              bi_vcnt;        /* how many bio_vec's       */
        unsigned short              bi_max_vecs;    /* max bvl_vecs we can hold */
        atomic_t                    bi_cnt;         /* pin count */
        struct bio_vec              *bi_io_vec;     /* the actual vec list */
        struct bio_set              *bi_pool;
        struct bio_vec              bi_inline_vecs[0];
};
```

```
struct bio_vec {
        struct page     *bv_page;
        unsigned int    bv_len;
        unsigned int    bv_offset;
 };
```

```
struct bvec_iter {
     sector_t       bi_sector;          /* device address in 512 byte sectors */
     unsigned int   bi_size;            /* residual I/O count */
     unsigned int   bi_idx;             /* current index into bvl_vec */
     unsigned int   bi_bvec_done;       /* number of bytes completed in
                                           current bvec */
};
```

The primary purpose of a bio structure is to represent an in-flight block I/O operation, as a list of segments.
A segment is a chunk of a buffer that is contiguous in memory.
Thus, individual buffers need not be contiguous in memory.
By allowing the buffers to be described in chunks, the bio structure provides the capability for the kernel to perform block I/O operations of even a single buffer from multiple locations in memory.
Vector I/O such as this is called scatter-gather I/O.

Fig. 4. Relationship between struct bio, struct bio_vec and struct page



IO VECTORS:

- The bi_io_vec field points to an array of bio_vec structures. Each bio_vec is treated as a vector of the form <page, offset, len>, which describes a specific segment: the physical page on which it lies, the location of the block as an offset into the page, and the length of the block starting from the given offset.
- In each given block I/O operation, there are bi_vcnt vectors in the bio_vec array starting with bi_io_vec. As the block I/O operation is carried out, the bi_idx field is used to point to the current index into the array.
- bi_iter of type struct bvec_iter , member of the bio structure has the starting sector (512 byte) at which the IO should start.
- bi_iter.bi_sector will have the starting sector for the IO.
- bi_bdev represents the target block device
- bi_end_io is the callback function, that gets called when the submitted bio has been completed

BIO FUNCTIONS:

- bio_alloc – allocate memory for a bio structure and get a reference.

  struct bio *bio_alloc(gfp_t gfp_mask, unsigned int nr_iovecs)

- bio_add_page – adds a page to the bio structure's bi_io_vec

  int bio_add_page(struct bio *, struct page *, unsigned int,unsigned int);

BIO submission in work handler function:

Work handler function will allocate bio structures with 64 pages. Linux supports upto 254 pages per bio.

Bio with 64 pages will yield good performance on all devices.

PAGE_SIZE – 4096 bytes

SECTOR SIZE – 512 bytes (used by the kernel)

BIO size – 64 pages

SEQUENTIAL READ and WRITE

Using the “io_load” structure member values, “devicesize” ,“runtime” and “pattern” are accessed.

Using the size of the target device, devicesize – in bytes

last_bio_sector - the last sector starting at which exactly a bio (64 pages) will fit into the device.

For sequential read or writes,

- Keep checking for runtime expiration
- allocate bio which can contain 64 pages
- Fill bio.bi_bdev – target block device
- Fill bio.bi_end_io – BIO end function callback
- Fill bio.bi_private - which will be used in the BIO end function callback
- Add 64 pages to the bio structure

    Use alloc_page() function to allocate page structure

    In case of writes,

        Use page_address() to get the address of the page in memory.

        Fill the buffer with random data or user-specified pattern

- Add the page to the BIO bi_io_vec vector
- Submit the bio to the block layer, after adding 64 pages
- Increment io_pending
- Reset bio_start_sector

    Bio_start_sector will be equal to bio_start_Sector + (64 *8)

    If end of the device has been reached, bio_start_sector will again be 0.

When the run time has expired, set “running” field to 0, to indicate that the work queue has stopped execution and that timer has been expired.

PSEUDOCODE

```
last_bio_sector  = devicesize – (PAGE_SIZE * 64)

bio_start_sector = 0

when (runtime has not expired)

        if bio_start_sector < last_bio_sector                /*Keep reading or writing till end of device*/

                count = 64;

                bio = bio_alloc(GFP_KERNEL, count) /* 64 io vectors, each representing a page)

                Fill in bio.bi_bdev                          /* device on which IO is generated */

                bio.bi_iter.bi_sector = bio_start_sector  /* device start sector in 512-byte*/

                bio.bi_end_io = custom_bio_end_function

                bio.bi_private = end      /* struct   check_end_io */

                for (count number of times)  /* 64 times */

                        page = alloc_page(GFP_NOIO)

                        buffer = page_address(page)              /* incase of write operation */

                        get_random_bytes(buffer, PAGE_SIZE)   /*incase of write operation */

                        bio_add_page(bio, page, PAGE_SIZE, 0)   /*add pages to IO vector */

                submit_bio(READ, bio)

                increment iopending member of “check_io_done” structure

                bio_start_sector = bio_start_sector + (count * 8)

        else                        /* Start over again */

                bio_start_sector = 0
```

RANDOM READ and WRITE

Using the “io_load” structure member values, “devicesize” ,“runtime” and “pattern” are accessed.

Using the size of the target device, devicesize – in bytes

For random read or writes,

- Keep checking for runtime expiration
- allocate bio which can contain exactly 1 page
- Fill bio.bi_bdev – target block device
- Fill bio.bi_end_io – BIO end function callback
- Fill bio.bi_private - which will be used in the BIO end function callback
- Add 1 page to the bio structure

    Use alloc_page() function to allocate page structure

    In case of writes,

        Use page_address() to get the address of the page in memory.

        Fill the buffer with random data or user-specified pattern

    Add the page to the BIO bi_io_vec vector

- Submit the bio to the block layer, after adding the page
- Increment io_pending
- Reset bio_start_sector

    Bio_start_sector will now be a random sector, which will be within the limits of the device size.

    Randomize_range() function is used to get the random address

    From the random address, random sector can be manipulated.

When the run time has expired, set “running” field to 0, to indicate that the work queue has stopped execution and that timer has been expired.

PSEUDOCODE

```
bio_start_sector = 0
last_page_sector - the last sector starting at which exactly one page will fit in the device
when (runtime has not expired)
        count = 1;
        bio = bio_alloc(GFP_KERNEL, count) /* 64 io vectors, each representing a page)
        Fill in bio.bi_bdev                     /* device on which IO is generated */
        bio.bi_iter.bi_sector = bio_start_sector  /* device start sector in 512-byte*/
        bio.bi_end_io = custom_bio_end_function
        bio.bi_private = end    /* struct   check_end_io */
        page = alloc_page(GFP_NOIO)
        buffer = page_address(page)             /* incase of write operation */
        get_random_bytes(buffer, PAGE_SIZE)   /*incase of write operation */
        bio_add_page(bio, page, PAGE_SIZE, 0)   /*add pages to IO vector */
        submit_bio(READ, bio)                       /*submit_bio(WRITE, bio) INCASE OF
WRITE */
        increment iopending member of "check_io_done" structure
        /* gets a random page-aligned              address */
        start_address = randomize_range(0, devicesize, devicesize)
        bio_start_sector = start_address >> 9   /* To get sector corresponding to the address */
```

BIO END FUNCTION CALLBACK

- This function will decrement “io_pending” field using the bi_private field, as the submitted BIO has been completed.
- Call complete function, to signal completion of the IO workload so that driver will return to the application with successful return code.
- Complete function will be called
    - If the work handler function has stopped execution

      That is, if runtime has expired

      and

    - if “io_pending” equals 0

      That is, if all submitted BIO’s have been completed

CONCLUSION

This kernel level IO tool generates IO in the form of BIO, and passes it on to the block layer, bypassing VFS layer and filesystem layer.

Bypassing these layers, will help to get closer to the device, and hence accurate measurement of IOPS and latency will be possible.