COMP3301 Assignment 3

Mass Storage and File Systems - Cache Me If You Can

Due: 1pm Monday in Week 13 (27th of October)
Submission: Git (code) and Blackboard (reflection)
Code submission is marked in your prac session in week 13
Last Updated: October 6, 2025

1 Academic Integrity

All assessments are **individual**. You should feel free to discuss aspects of C programming and assessment specifications with fellow students and discuss the related APIs in general terms. You should not actively help (or seek help from) other students with the actual coding of your assessment. It is cheating to look at another student's code, and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code will be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion (outside of the base code given to everyone), formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a teaching staff member. Do not be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school website: https://eecs.uq.edu.au/current-students/guidelines-and-policies-students/student-conduct.

Do not post your code to a public place such as the course discussion forum or a public code repository. (Code in private posts to the discussion forum is permitted.) You must assume that some students in the course may have very long extensions so do not post your code to any public repository. You must follow the following code usage and referencing rules for all code committed to your Git repository (not just the version that you submit), in Table 1.

Table 1: Code Origin and Uses/References

Code Origin	Usage/Referencing
Code provided by teaching staff this semester Code provided to you by COMP3301 teaching staff or posted on the discussion forum by teaching staff.	Permitted May be used freely without reference. (You must be able to point to the source if queried about it - so you may find it easier to reference the code.)
Code you wrote this semester for this course Code you have personally written this semester for COMP3301.	Permitted It may be used freely without reference, provided you have not shared or published it.
Unpublished code you wrote earlier Code you have personally written in a previous enrollment in this course or in another UQ course or for other reasons and where that code has not been shared with any other the person or published in any way or submitted for assessment.	Conditions apply; requires references May be used provided you understand the code AND the source of the code is referenced in a comment adjacent to that code. If such code is used without appropriate referencing, then this will be considered misconduct.
C Code from AI tools	Conditions apply; requires references

Whilst students may use AI and/or MT technologies, successful completion of the assessment in this course will require students to critically engage in specific contexts and tasks for which artificial intelligence will provide only limited support and guidance.	**A failure to reference generative AI or MT use may constitute student misconduct under the Student Code of Conduct.** To pass this assessment, students will be required to demonstrate detailed comprehension of their submission, independent of AI and MT tools.
Code copied from other sources Code, in any programming language: • copied from any website or forum (including Stack Overflow and CSDN); • copied from any public or private repositories; • copied from textbooks, publications, videos, apps; • copied from code provided by teaching staff only in a previous offering of this course; • written by or partially written by someone else or written with the assistance of someone else (other than a teaching staff member); • written by an AI tool that you did not personally and solely interact with; • written by you and available to other students; or • from any other source besides those mentioned in earlier table rows above.	Prohibited May not be used. If the source of the code is referenced adjacent to the code then this will be considered code without academic merit (not misconduct) and will be removed from your assignment prior to marking (which may cause compilation to fail and zero marks to be awarded). Copied code without adjacent referencing will be considered misconduct and action will be taken. This prohibition includes code written in other programming languages that has been converted to C.
Code that you have learned from Examples, websites, discussions, videos, code (in any programming language), etc. that you have learned from or that you have taken inspiration from or based any part of your code on but have not copied or just converted from another programming language. This includes learning about the existence of and behaviour of library functions.	Conditions Apply, references required May be used provided you do not directly copy code AND you understand the code AND the source of the code or inspiration or learning is referenced in a comment adjacent to that code. If such code is used without appropriate referencing then this will be considered misconduct.

2 Introduction

In this assignment, you will be implementing a block device driver to provide support for a virtual disk format called FVD (Forkable Virtual Disk) in OpenBSD. A minimal block device template is given to you in the form of a base code patch.

This assignment aims to strengthen your understanding of block devices, mass storage and filesystems covered in the week 6, 8 and 9 lectures, as well as main/virtual memory concepts like copy-on-write and cache replacement covered in the week 5 lecture. It also assesses your skills in reading a file format specification and implementing code accordingly.

3 Background

This assignment extends the OpenBSD kernel to add support for using FVD disk images as block devices. This is similar to the existing functionality provided by the `vnd(4)` driver, which supports using files containing a disk image as a block device.

From a high-level point of view, a physical disk device presents a sequence of bytes that can be written to or read from, with the ability to quickly seek to an arbitrary position and read and write at that point. Note that this is a simplification that ignores that disks address and provide access to blocks of bytes, not individual bytes.

A file on most operating systems offers similar features, i.e., a sequence of bytes that can be accessed by address. Because of these similarities, it is possible for an operating system to provide a common set of operations on both files and disks (e.g., open, close, read, write, seek, etc.) and allow them to be used interchangeably. For example, you could use `tar(1)` to write an archive to a file in a filesystem, or directly to a disk device. `dd(1)`, `cp(1)`, `cat(1)`, etc can read the bytes from a raw disk into a file, and vice versa. However, operating systems generally provide extra functionality on top of disk devices such as the ability to partition disks and mount filesystems from them.

3.1 `vnd(4)`

The `vnd(4)` driver in OpenBSD provides a "disk-like interface to a file". This means the OpenBSD kernel can open a file and present it as a block device to the rest of the system, which in turn allows for the creation and use of filesystems on these disk images.

The `vnd(4)` driver currently only supports using raw disk images as backing files. There's a one-to-one mapping of data offsets for data in the virtual disk device and the byte offset of that data in the underlying file. This makes the implementation very simple, with the downside that the backing file has to be the same size as the disk `vnd(4)` is presenting. If you have a 32G disk image, the file will be 32G regardless of how much data is actually stored inside a filesystem mounted on top of it. Similar functionality exists in the `loop(4)` driver in Linux, and the `lofi(4)` driver in Solaris and Illumos.

3.2 On-Demand Space Allocation

On-demand space allocation can be used with disk images, so that the backing file starts off being effectively empty, and grows in size as you write to it. For example, a virtual 32G disk could start off with a backing file size of 10M (for storing metadata), and if you only write 8GB of data to the virtual disk, the size of the backing file becomes 8G rather than 32G. This is useful for saving space, especially when transferring disk images over the network. Similar concepts exist in memory overcommit demand paging.

3.3 State Forking and Copy-on-Write

The ability to fork the state of a disk, effectively turning it into multiple identical disks (branches) is useful, for both backup and isolation purposes. Similar concepts of forking and branching can be found in Git - the version control software used in COMP3301.

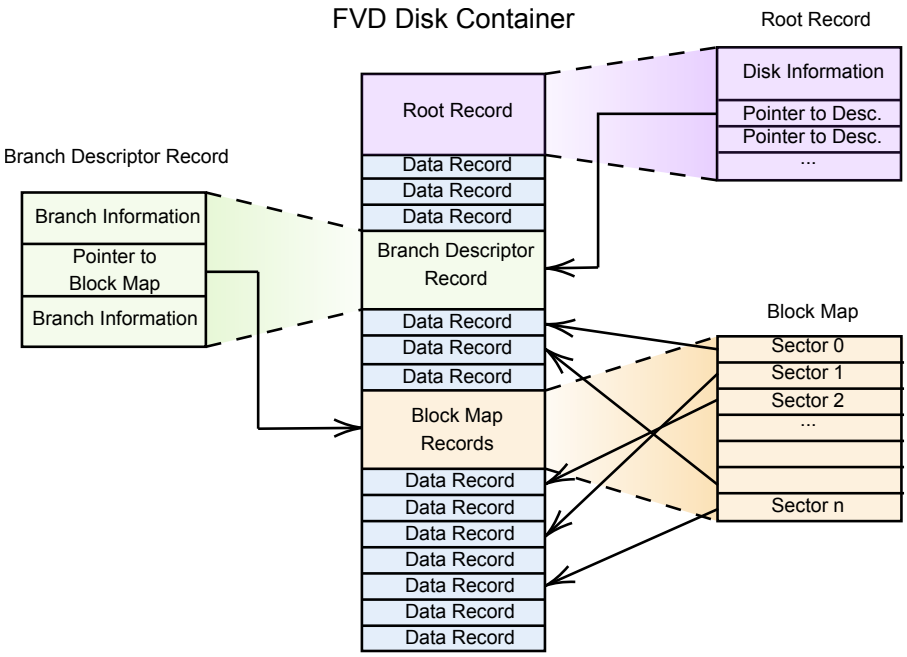
Given that the state of a branch is identical to its parent when when initially forked, copy-on-write can be used rather than making full copies of the disk state, to reduce storage use. Copy-on-write essentially means to not actually duplicate the data even though the user asked for it to be duplicated, and instead duplicate it only when the user tries to modify it. The same concept is used when you call `fork(2)` to fork a process - memory between the parent and child is shared and no new memory is allocated until one modifies a shared page.

3.4 Forkable Virtual Disk (FVD)

Forkable Virtual Disk Image (FVD) is a virtual disk image format designed by the COMP3301 teaching team while being chased by an angry mob of students holding pitchforks (and they say to the students - *cache* me if you can). The two defining features of FVD are that it allocates space on demand, and that it allows for the disk state to be forked at any point into separate branches. These features make it more flexible and usable than raw disk images used by `vnd(4)`.

An FVD image contains 2 files - the disk container and refcount metadata. The disk container file has extension `.fvd` and is comprised of fixed-size records. The refcount metadat file has extension `.fvd.ref` and is comprised of unsigned 8-bit integers, where each hold the refcount of the corresponding record in the disk container.

FVD achieves on-demand space allocation and copy-on-write using block maps. A block map maps all sectors within the virtual disk to a record inside the disk image. Sectors which have never been written to are not implicitly mapped to any existing record, but instead use the special record number `0` to indicate that they are empty and contain all zeros. The concept of block maps which map sectors to records is similar to the concept of page tables, which map virtual pages to physical frames.



A specification of the FVD format may be accessed through Blackboard or using the following link: https://stluc.manta.uqcloud.net/comp3301/public/a3-fvd-spec.pdf.

4 Instructions

To complete the assignment, you will need to do the following:

1. Download the base code patch

```
$ cd ~
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/a3-base.patch
```

2. Create the `a3` branch and apply base code patch

```
$ cd /usr/src
$ git checkout -b a3 base
$ git am < ~/a3-base.patch
```

3. Build and install the updated kernel

```
$ cd /usr/src/sys/arch/amd64/compile/GENERIC.MP
$ make config
$ make -j4
$ doas make install
$ doas reboot
```

4. Build and install the updated includes

```
$ cd /usr/src/include
$ doas make includes
```

5. Build and install `fvdctl(8)`

```
$ cd /usr/src/usr.sbin/fvdctl
$ make obj
$ make
$ doas make install
```

6. Create `fvd(4)` device nodes

```
$ doas cp /usr/src/etc/etc.amd64/MAKEDEV /dev
$ cd /dev
$ doas chmod 755 MAKEDEV
$ doas ./MAKEDEV fvd0 fvd1 fvd2 fvd3
```

7. Install `fvdtool(1)`

```
$ curl https://stluc.manta.uqcloud.net/comp3301/public/fvdtool_ins\
  tall | doas sh -s --
```

5 Assignment Tasks

You will be extending the OpenBSD kernel to add support for using FVD images as backends for fvd(4) virtual block devices. fvd(4) is roughly based on vnd(4). Boilerplate code for the device entry points and some features have been provided in the base code, you are only required to implement the read/write functionality and the IOCTLs listed below.

All IOCTLs listed in this section should only work on the raw partition (partition c) of the character device (/dev/r*) associated with each fvd(4) disk (e.g., /dev/rfvdXc). Except for FVDIOC_ATTACH, it should only work when an fvd(4) disk is attached to a backing file.

5.1 FVD Attach/Detach and Info

Before a FVD image can be used, it must be attached to an fvd(4) node. Allow attaching to an fvd(4) node by implementing the FVDIOC_ATTACH IOCTL:

FVDIOC_ATTACH

Specify the FVD image to attach, and the branch to use. This IOCTL takes in an argument consisting of the following structure to be used for input:

```
struct fvd_attach {
    const char  *fa_path;       /* path to FVD image */
    const char  *fa_branch;     /* branch to attach */
    int          fa_readonly;   /* whether to attach as read-only */
};
```

fa_path The path of the FVD image to attach a fvd disk to. This path is the path to the FVD disk container file.

fa_branch The name of the branch within the FVD image to attach to, empty string means to attach to the default branch.

fa_readonly Whether to allow writing to the virtual disk after attaching, zero means writable and non-zero means read-only. An FVD image which you do not have write access to can only be attached as read-only.

This IOCTL should fail with with an appropriate errno(2) error number if the backing file is not a valid FVD image (determined from examining the Root Record and the Branch Descriptor Record of the branch to attach), and with ESRCH if the branch does not exist within the backing file.

Attached FVD images should also be detachable. The following detach IOCTL is implemented for you already, and listed here for documentation purposes:

FVDIOC_DETACH

Requests for the FVD image to be detached and backing files (`.fvd` and `.fvd.ref`) closed. This `IOCTL` takes in an integer argument which acts as a boolean, for indicating whether it is a forced detach.

This `IOCTL` fails with `EBUSY` if the virtual disk is currently in use, unless the `IOCTL` argument is non-zero, in which case the virtual disk is forcefully detached. `ENXIO` is returned if there is no FVD image attached.

Hint: While this `IOCTL` is already implemented for you, if you allocate other resources on attach, you also need to free them on detach. The actual closing of the `.fvd` and `.fvd.ref` files and deallocation of resources happen in `fvd_leave()`, not in `fvd_detach()` or `fvd_close()`.

`FVDIOC_INFO`

Requests the name of the FVD file used for the currently attached `fvd(4)` block device and the currently attached branch. This `IOCTL` takes an argument consisting of the following structure to be used for output:

```
struct fvd_info {
    char        fi_path[1024];          /* path to attached FVD image */
    char        fi_branch[FVD_MAX_BNM]; /* current branch name */
};
```

`fi_file` The path of the attached FVD image, identical to what was specified as `fa_file` in the `FVDIOC_ATTACH IOCTL`. Only the first 1023 characters need to be returned. This string is `NUL`-terminated.

`fi_branch` The name of the currently-attached branch. If the default branch is attached, it could either be the empty string or the actual name of the default branch. This string is NUL-terminated.

This `IOCTL` should always succeed as long as an FVD image is attached, otherwise fail with `ENXIO`.

5.2 Virtual Disk I/O

Implement the `fvd_read_sect()` and `fvd_write_sect()` functions to allow reading from and writing to the current branch of the attached virtual disk. The virtual disk is accessed through the device nodes (special files) `/dev/{,r}fvd*`. The node `/dev/rfvdXc` (where `X` is the unit number) is a raw representation of the entire virtual disk as a file, and can be read from and written to like a regular file.

```
static int
fvd_read_sect(struct fvd_softc *sc, uint32_t sec, void *buf);

static int
```

```
fvd_write_sect(struct fvd_softc *sc, uint32_t sec, const void *buf);
```

sc The `softc` (software context) of the attached FVD.
sec The sector number of the sector to read/write.
buf The buffer where the sector is read to or written from.

These functions should return `0` on success or an `errno(2)` error number on failure.

Writing to a sector which had never been written to before (hence not actually allocated in the FVD image) should result in that sector being allocated. Writing to a sector which maps to a record shared by multiple branches should result in a copy-on-write operation, such that the original record is left unmodified and a new record is allocated and written to, with the entry for that sector in the block map updated to point to the newly-allocated record.

Hint: Reads and writes on disk container and refcount metadata files can be carried out using `vn_rdwr()`.

If your implementation is correct, then it should be possible to initialise the disk with a GPT partition table using `fdisk(8)`, create partitions with `disklabel(8)` and initialise filesystems on the partitions using `newfs(8)`. Afterwards, filesystems on the partitions may be mounted and unmounted using `mount(8)` and `umount(8)`. See example commands and outputs in the Appendix 1: Filesystem Example.

5.3 Virtual Disk State Forking

A crucial feature of FVD over raw disk images is its ability to fork a virtual disk's state at any point. Implement the following `IOCTL`:

`FVDIOC_FORK`

Forks the state of the current branch of the attached FVD image and immediately switch to the new branch. The newly-forked branch should initially be identical to the previous branch content-wise. Any subsequent reads/writes should be applied to the newly forked branch. This `IOCTL` takes an argument consisting of the following structure to be used for input:

```
struct fvd_fork {
    char    ff_name[FVD_MAX_BNM];   /* new branch name */
    int     ff_force;               /* forced fork */
};
```

`ff_name` The name of the new branch. This string is `NUL`-terminated.

`ff_force` Whether to perform the fork forcefully even if the virtual disk is in use (has a filesystem mounted).

This `IOCTL` should fail with `EEXIST` if the branch name specified is empty or already exists.

If the attached disk is current in use and the `ff_force` argument is zero, `EBUSY` is returned instead of performing the fork - this is done to prevent filesystem corruption on the parent branch.

Forking the disk state should not cause duplication of data records, instead copy-on-write should be used. This means the same sector in both the parent and the child branch should point to the same record, as long as neither are modified. Incrementing the refcount of records shared by both the parent and the child is necessary.

5.4 Sector Caching and Cache Replacement

Due to the high latency of disk I/O, operating systems often cache a certain number of sectors for each disk in memory to improve performance. In this assignment, you are asked to cache 16 sectors for each `fvd(4)` device. Only sectors of the virtual disk are cached (not FVD records, though you may cache those too separately if you wish).

Note: In real life, you would be caching a lot more sectors than 16, often in the order of megabytes of data, however for the simplify of implementation and testing, we ask you to only cache 16 sectors (8 KiB). Your solution, however, should be portable such that the cache size can be adjusted easily.

For example, if you read a sector of the disk, you should cache it and not need to read that sector from the FVD image again until it is replaced. The same goes for writes, when a sector is written for the first time, the write is cached (not written immediately) and further reads and writes on the same sector should not result in an I/O on the FVD image until when it is removed from the cache, at which time it is written to the FVD image.

Given that caching disk sectors is a lot like demand paging, the page replacement algorithms covered in the week 5 lecture *Main and Virtual Memory* can also be used for replacing cached sectors. In this assignment, you are asked to use the **Least Recently Used (LRU)** algorithm.

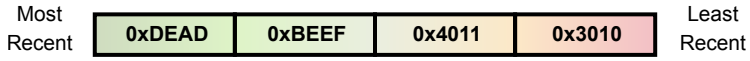
LRU Cache Replacement Example

For the sake of simplicity, this example uses a cache size of 4. Sectors `0x3301`, `0x3010`, `0x4011`, `0xBEEF`, `0xDEAD`, `0x4011` and `0x1234` are accessed. Initially, the cache is empty:

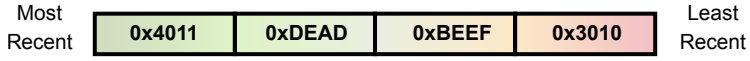
Sector `0x3301` is accessed:

Then sectors `0x3010`, `0x4011` and `0xBEEF` are accessed:

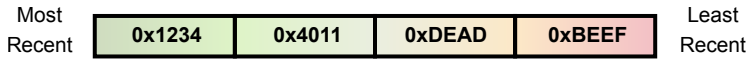
At this point, the entire cache is full, with `0xBEEF` being the most recently accessed sector and `0x3301` being the least recently accessed sector. Sector `0xDEAD` is then accessed:



The least recently used sector `0x3301` is replaced, with changes written to the FVD image if it was dirty. Next, the sector `0x4011` is accessed:



Nothing has been removed from or added to cache, however now sector `0x4011` is the most recently accessed sector. Last, sector `0x1234` is accessed:



Sector `0x1234` becomes the most recently accessed sector and sector `0x3010` is removed from cache (with changes flushed).

In order to test your implementation of the cache replacement algorithm, you are to implement the following `IOCTL`s:

`FVDIOC_CACHE_LIST`

Lists the sectors currently in cache, and their status. This `IOCTL` takes an argument consisting of the following structure to be used for output:

```
struct fvd_cache_list {
    size_t          fc_nsects;      /* number of sectors in cache */
    struct sec_info fc_sects[16];   /* info about cached sectors */
};

struct sec_info {
    uint32_t        si_number;      /* sector number */
    uint32_t        si_checksum;    /* sector checksum */
    int             si_dirty;       /* whether sector is dirty */
};
```

`fc_nsects`	The number of sectors currently in cache.
`fc_sects`	Array of cached sectors. Ordering of elements in this array does not matter, however only the first `fc_nsects` elements of this array should be populated.
`si_number`	The sector number of the sector in cache.
`si_checksum`	The result of calling `fvd_csum_sect()` on the cached sector data.
`si_dirty`	Whether the sector in cache is dirty (modified).

This `IOCTL` should fail with `ENXIO` if no FVD image is attached, otherwise it should always succeed.

`FVDIOC_CACHE_EMPTY`

Removes all sectors from the cache, flushing any dirty (modified) ones to the FVD image file. This `IOCTL` takes no arguments.

This `IOCTL` should fail with an appropriate `errno(3)` error number if writing of any dirty sectors fail, otherwise it should always succeed.

When a fork is performed, all cached sectors which are dirty (modified) are flushed (you don't have to empty the cache, though you can if you wish). The same goes with when the FVD image is detached, except in this case the cache must be emptied.

5.5 Reflection

Consider the following scenario:

> Puffy, a hypothetical user of your driver, had accidentally deleted the refcount metadata (`.fvd.ref`) file of their most important FVD image, and now could no longer use it. As a result, Puffy had lodged a feature request for the FVD format to store the actual records and refcounts in a single file. Puffy had also noticed that the block maps take up too much space and would appreciate a solution to reduce its size.

Reflect on your understanding of course content and this assignment by answering the following questions:

1. Can Puffy still recover the data from their FVD image using only the `.fvd` file? Explain why or why not.
2. Would contiguous allocation (covered in week 9 lecture) be suitable for the storage of the refcount metadata within the FVD file? Explain why or why not.
3. What are the disadvantages of using linked allocation (covered in week 9 lecture) for storing the refcount metadata within the FVD file? List at least one.
4. How would you reduce the size of the block maps? List one disadvantage associated with it.

These questions are open-ended, there is no expected answer. Upload your reflection as a PDF file to the Blackboard A3 reflection submission. Page length is a maximum of 2 pages or less. **PDF name must be STUDENT_NUMBER_A3.pdf.** Note this is your `4XXXXXXX` ID number and not your `s4XXXXXX` login.

6 Provided Tools/Files

6.1 `fvdctl(8)`

The patch includes source code for `fvdctl(8)`, a utility which uses the `IOCTL`s defined previously to control `fvd(4)` devices. It is similar to `vnconfig(8)`. The source can be found after the patch is applied in `src/usr.sbin/fvdctl`.

`fvdctl attach`

The `attach` sub-command attaches an FVD image to an `fvd(4)` node. It takes the following arguments:

```
fvdctl attach [-Dr] fvdX image.fvd [branch]

  -D          Treats the fvdX argument as a path to a device special file rather than
              the name of the fvd(4) node.
  -r          Attaches the FVD image as read-only.
  fvdX        Name of the fvd(4) node, e.g., fvd0, fvd1.
  image.fvd   Path to the FVD image.
  branch      Name of the branch to attach, or the default branch if not supplied.
```

`fvdctl detach`

The `detach` sub-command detaches an FVD image from an `fvd(4)` node. It takes the following arguments:

```
fvdctl detach [-Df] fvdX

  -D    Treats the fvdX argument as a path to a device special file rather than
        the name of the fvd(4) node.
  -f    Forcefully detaches the virtual disk even if it is currently in use.
  fvdX  Name of the fvd(4) node, e.g., fvd0, fvd1.
```

`fvdctl info`

The `info` sub-command prints the path to and branch of the FVD image attached to an `fvd(4)` node. It takes the following arguments:

```
fvdctl info [-D] fvdX

  -D    Treats the fvdX argument as a path to a device special file rather than
        the name of the fvd(4) node.
  fvdX  Name of the fvd(4) node, e.g., fvd0, fvd1.
```

`fvdctl fork`

The `fork` sub-command forks the current branch of the FVD image attached to an `fvd(4)` node, and switches to that branch immediately. It takes the following arguments:

```
fvdctl fork [-Df] fvdX name
```

`-D`	Treats the `fvdX` argument as a path to a device special file rather than the name of the `fvd(4)` node.
`-f`	Forcefully forks the current branch even if it is currently in use.
`fvdX`	Name of the `fvd(4)` node, e.g., `fvd0`, `fvd1`.
`name`	Name of the new branch after forking.

`fvdctl io`

The `io` sub-command allows interactive or scripted I/O to be performed on the virtual disk attached to an `fvd(4)` node. It takes the following arguments:

```
fvdctl io [-D] fvdX [script]
```

`-D`	Treats the `fvdX` argument as a path to a device special file rather than the name of the `fvd(4)` node.
`fvdX`	Name of the `fvd(4)` node, e.g., `fvd0`, `fvd1`.
`script`	Path to a script file. `stdin` is used if omitted.

The script is executed on a line-by line basis, with each line containing a command to execute. The first character of a line is the command opcode, followed by a whitespace-delimited list of arguments. The following commands are valid:

`r <sector>`	Reads a sector and prints checksum.
`w <sector> <char>`	Writes a sector filled with <char> and prints checksum.
`e`	Empties the sector cache.
`l`	Lists the sectors currently in cache.
`q`	Quits.
`#`	This line is a comment.

6.2 `fvdtool(1)`

You are provided with a tool called `fvdtool(1)` to create and manipulate FVD images. It is capable of:

- Creating FVD images (empty or from raw image)
- Displaying information about FVD images
- Exporting FVD branches to raw images
- Forking any branch of an FVD image
- Reading sectors from FVD images
- Writing sectors to FVD images

`fvdtool create`

The `create` sub-command creates an FVD image of a specified size. It takes the following arguments:

```
fvdtool create -s C,H,S [-r raw_img] fvd_img
```

-s C,H,S	The CHS (cylinder-head-sector) geometry of the virtual disk image to create.
-r raw_img	Create the FVD image based off the raw image `raw_img`.
fvd_img	The FVD image to create.

fvdtool info

The `info` sub-command displays the information of an FVD image and draws a branch tree:

```
fvdtool info fvd_img
```

fvd_img	The FVD image to display info of.

fvdtool export

The `export` sub-command exports an FVD image to a raw image. It takes the following arguments:

```
fvdtool export [-b branch] fvd_img raw_img
```

-b branch	The branch in the FVD image to export, or `default` if omitted.
fvd_img	The input FVD image.
raw_img	The output raw image.

fvdtool fork

The `fork` sub-command forks a branch of an FVD image. It takes the following arguments:

```
fvdtool fork [-b branch] fvd_img new_brch
```

-b branch	The branch in the FVD image to fork, or `default` if omitted.
fvd_img	The FVD image.
new_brcn	The name of the newly forked branch.

fvdtool read

The `read` sub-command reads a certain number of sectors from a particular branch of an FVD image:

```
fvdtool read [-b branch] fvd_img sec seccnt [out_file]
```

-b branch	The branch in the FVD image to read from, or `default` if omitted.
fvd_img	The FVD image to read from.
sec	The sector to start reading at.
seccnt	The number of sectors to read.
out_file	The file to output those sectors to. If omitted, the sectors are hex-dumped to `stdout`.

`fvdtool write`

The `write` sub-command writes a certain number of sectors to a particular branch of an FVD image:

```
fvdtool write [-b branch] fvd_img sec [in_file]
```

-b branch	The branch in the FVD image to write to, or `default` if omitted.
fvd_img	The FVD image to write to.
sec	The sector to start writing at.
in_file	The file to containing the sectors to write. If omitted, `stdin` is used. If the data is not a multiple of sector size, it is `NUL`-padded to the next sector boundary.

7 Misc. Requirements

7.1 Code Style

Your code is to be written according to OpenBSD's style guide, as per the style(9) man page. An automatic tool for checking for style violations is available at https://stluc.manta.uqcloud.net/comp3301/public/cstyle.pl.

Code style marks will be calculated based on the number of style violations in the code which you have written yourself or modified - style violations in the OpenBSD source tree or in the base code will not affect code style marks. Some level of functionality is required to score marks for code style (i.e., no submission implies no style violations, however no style marks will be awarded in that case).

7.2 Reliability, Robustness, Portability and Modularity

In order to score higher marks, your code is expected to be reliable and robust, that is it should handle all errors appropriately and should not crash unexpectedly. Your code should also be portable and modular, in the sense that constants should not be hard coded and similar code should not be duplicated in multiple areas. Your code should also be free of race conditions and undefined behaviour.

7.3 Compilation (Pass/Fail)

Your code must be compile-able under the `GENERIC.MP` (generic multiprocessor) configuration for the AMD64 (aka x86-64 and x64) architecture. Code with compile-time errors will be

marked manually, and may result in heavy penalties up to the deduction of all functionality marks.

8 Git Submission

Submission must be made electronically by committing and pushing your changes to your course-provided Git repository on `source.eait.uq.edu.au`. Code checked into any other branch in your repository or not pushed to the `a3` branch (i.e., left on your VM) will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code and makefiles. Please do not commit `cscope` outputs, core dumps or base code patch files.

Your `a3` branch should consist of:

- The OpenBSD base commit (provided)
- The A3 base patch commit (provided)
- Commit(s) for adding the required functionality (by yourself)

Commit history and commit messages do not contribute to your grade. However, it is strongly recommended that you commit frequently and use appropriate commit messages.

8.1 Marking

Your submission will be marked by course staff during an in-person demo with you at your prac session of the due week. You must attend your session in person; otherwise, your submission will not be marked. Online attendance (e.g., via Zoom) is not permitted.

8.2 Demo Session

You will be asked to demonstrate your assignment as part of your regular practical session. Demonstrations will run on a marking VM (not your VM), which has nothing but the latest code which you have pushed to the `a3` branch before the submission deadline. Manual and automated tests and manual code examinations will be conducted during the demo. Changes to your code during the session will not be accepted.

It is your responsibility to ensure that your code compiles and operates correctly. Code that fails to compile or code that crashes the kernel upon boot results in your submission not being marked for functionality.

You may be asked to explain certain aspects of your submission during the demo session. Most of the questions will be open-ended and have no single correct answer. However, failure to demonstrate your understanding of the code which you have presumably written or a wrong understanding of the concepts yet correct code may result in allegations of academic misconduct.

9 Recommendations

9.1 Backups

It is *highly recommended* that you use Git to regularly backup your assignment code. Beware that corruptions to your filesystem can happen in the event of kernel crashes, and will require you to start from scratch if there is no backup. Regularly committing code to Git and pushing to origin ensures that your code will not be lost in the event of a filesystem corruption.

It is also recommended that you take a snapshot of your virtual machine (VM) before you attempt the programming aspect of this assignment. This can be done by running `snapshot <name>` in your VM Control.

9.2 Relevant Manual Pages

The following manual pages may be useful for the understanding of existing functionalities or the implementation of the assignment. You are not required or expected to use all of these in the code which you write.

- vnd(4) - `vnode` Disk Driver
- vnode(9) - `vnodes`
- VNSUBR(9) - High-level Convenience Functions for `vnode` Operations
- VOP_LOOKUP(9) - `vnode` Operations
- vfs(9) - Kernel Interface to File Systems
- ioctl(2) - Control Device
- MAKEDEV(8) - Create System and Device Special Files
- malloc(9) - Kernel Memory Allocator
- queue(3) - Kernel List/Queue Macros
- TAILQ_INIT(3) - Doubly-Linked List Macros
- RB_PROTOTYPE(3) - Red-Black Tree Macros
- KASSERT(9) - Kernel Assert Routines
- rwlock(9) - Read/Write Lock

9.3 Testing

Public tests are provided for this assignment. It is suggested that you run the command `sync` before running any of the tests (or any test program which you write), to minimise the chance of filesystem corruption in the event of an unexpected kernel crash.

The public tests only test the very basic functionalities of your implementation, and therefore your implementation is not guaranteed to be fully correct even if you pass all the public tests. You are encouraged to write your own test cases.

Installation

The public tests can be installed by running the following commands in your VM:

```
$ cd ~
$ ftp https://stluc.manta.uqcloud.net/comp3301/public/a3_public_tests.tar
$ tar -xvf a3_public_tests.tar
$ cd a3test
$ ./install
```

This will install the test program `a3test` and 24 public tests.

`a3test` Test Program

The `a3test` program takes the following command-line arguments:

```
$ a3test <public|custom|all> <run|info|dump> <all|name,name2...>
```

The first argument must be one of `public`, `custom` or `all`. `public` tells the test program to search for public tests, `custom` tells the test program to search for custom tests (which you can write yourself) and `all` tells the test program to search for all tests.

The second argument must be one of `run`, `info` or `dump`. `run` executes the selected test(s), `info` prints out the information about the selected test(s) and `dump` dumps your outputs (`stdout` and `stderr`) and the expected outputs of the selected test(s) to the current directory (in case you want to run `diff(1)` or `sdiff(1)`).

The third argument is a list of tests to run/info/dump, or `all` to run/info/dump all tests. The list is comma separated.

Example Test Commands

To run:

- All test, you run `a3test all run all`.
- A particular test (e.g., 3.2), you run `a3test all run 3.2`.
- A selection of tests (e.g., 1.5, 3.2 and 4.1), you run `a3test all run 1.5,3.2,4.1`.

To display the information about:

- All tests, you run `a3test all info all`.
- A particular test (e.g., 1.6), you run `a3test all info 1.6`.
- A selection of tests (e.g., 1.3, 2.1 and 3.1), you run `a3test all info 1.3,2.1,3.1`.

To dump your outputs to the current directory for:

- All tests, you run `a3test all dump all`.
- A particular test (e.g., 1.4), you run `a3test all dump 1.4`.
- A selection of tests (e.g., 1.1, 4.1, 5.1), you run `a3test all dump 1.1,4.1,5.1`.

Examples of passed and failed tests with screenshots are in the A3 Tests thread on Ed.

10 Specification Clarifications

Specification clarifications and/or updates may be issued, they will be communicated through Blackboard and/or Ed. If needed, you are encouraged to seek for spec clarifications in the A3 Spec Clarification Megathread on Ed rather than by making individual posts or emailing teaching staff.

All clarifications made since the initial release of this specification are coloured red in this document.

11 Appendix

Appendix 1: Filesystem Example

> **Note:** This is just an example to give you an idea of how to play around with filesystems. The output (such as timestamps and file sizes) don't have to match exactly.

```
uqygao13:~$ fvdtool create -s 514,16,255 1gb.fvd
Creating FVD with the following parameters:
  [+] disk container: "1gb.fvd"
  [+] refcount metadata: "1gb.fvd.ref"
  [+] 1023.98 MiB - 2097120 sectors (CHS = 514,16,255)

Disk image successfully created!
uqygao13:~$ ls -l 1gb.fvd
-rw-r--r--  1 uqygao13  uqygao13  8389632 Oct  6 19:22 1gb.fvd
uqygao13:~$ doas fvdctl attach fvd0 1gb.fvd
uqygao13:~$ doas fdisk -g fvd0
Do you wish to write new GPT? [n] y
Writing GPT.
uqygao13:~$ doas fdisk fvd0
Disk: fvd0       Usable LBA: 34 to 2097086 [2097120 Sectors]
   #: type                                  [       start:         size ]
-------------------------------------------------------------------------
   0: OpenBSD                               [          64:      2097023 ]
uqygao13:~$ doas disklabel -dE fvd0
Label editor (enter '?' for help at any prompt)
fvd0> p
OpenBSD area: 64-2097087; size: 2097023; free: 2097023
#                size           offset  fstype [fsize bsize   cpg]
  c:          2097120                0  unused
fvd0> a
partition to add: [a] ↩
offset: [64] ↩
size: [2097023] ↩
FS type: [4.2BSD] ↩
```

```
fvd0*> w
fvd0> q
No label changes.
uqygao13:~$ doas disklabel fvd0
# /dev/rfvd0c:
type: vnd
disk: FVD Image
label: comp3301
duid: adca42d176f24d9a
flags:
bytes/sector: 512
sectors/track: 255
tracks/cylinder: 16
sectors/cylinder: 4080
cylinders: 514
total sectors: 2097120
boundstart: 64
boundend: 2097087

16 partitions:
#                size           offset  fstype [fsize bsize   cpg]
  a:          2096992               64  4.2BSD   2048 16384     1
  c:          2097120                0  unused
uqygao13:~$ ls -l 1gb.fvd
-rw-r--r--  1 uqygao13  uqygao13  8424448 Oct  6 19:24 1gb.fvd
uqygao13:~$ doas newfs -q /dev/rfvd0a
/dev/rfvd0a: 1023.9MB in 2096992 sectors of 512 bytes
6 cylinder groups of 202.50MB, 12960 blocks, 25920 inodes each
uqygao13:~$ ls -l 1gb.fvd
-rw-r--r--  1 uqygao13  uqygao13  8830464 Oct  6 19:24 1gb.fvd
uqygao13:~$ doas mount /dev/fvd0a /mnt
uqygao13:~$ doas chown uqygao13 /mnt
uqygao13:~$ cd /mnt
uqygao13:/mnt$ git clone https://github.com/DoctorWkt/pdp7-unix
Cloning into 'pdp7-unix'...
remote: Enumerating objects: 4599, done.
remote: Counting objects: 100% (71/71), done.
remote: Compressing objects: 100% (45/45), done.
remote: Total 4599 (delta 28), reused 53 (delta 25), pack-reused 4528 (from 1)
Receiving objects: 100% (4599/4599), 1.34 MiB | 288.00 KiB/s, done.
Resolving deltas: 100% (2885/2885), done.
Updating files: 100% (249/249), done.
uqygao13:/mnt$ rm -rf pdp7-unix/.git
uqygao13:/mnt$ find .
.
./pdp7-unix
./pdp7-unix/.github
```

```
./pdp7-unix/.github/workflows
./pdp7-unix/.github/workflows/main.yml
./pdp7-unix/.gitignore
./pdp7-unix/LICENSE
./pdp7-unix/Makefile
./pdp7-unix/README.md
./pdp7-unix/build
./pdp7-unix/build/Makefile
./pdp7-unix/build/Notes.md
./pdp7-unix/build/alt
        ...
./pdp7-unix/tools/README.md
./pdp7-unix/tools/a7out
./pdp7-unix/tools/as7
./pdp7-unix/tools/b.c
./pdp7-unix/tools/ccov7
./pdp7-unix/tools/compare
./pdp7-unix/tools/fsck7
./pdp7-unix/tools/mkfs7
./pdp7-unix/tools/sdump
./pdp7-unix/tools/xref7
uqygao13:/mnt$ cd ~
uqygao13:~$ doas umount /mnt
uqygao13:~$ doas fvdctl detach fvd0
uqygao13:~$ ls -l 1gb.fvd
-rw-r--r--  1 uqygao13  uqygao13  11916800 Oct  6 19:35 1gb.fvd
```