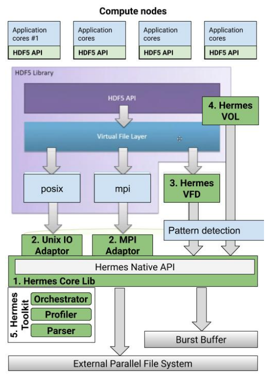
Adapters

Instead of the Hermes primitives, existing applications use I/O libraries and middleware such as the C standard I/O library, MPI-IO, or HDF5. Since Hermes is intended to be a seamless I/O buffering solution, a set of *adapters* is provided in the form of `LD_PRELOAD`-able shared libraries. These adapters perform the task of mapping between the application view of I/O primitives (e.g., files) and the Hermes primitives. An exemplary discussion of such mappings can be found in *Enosis: Bridging the Semantic Gap between File-based and Object-based Data Models* (http://www.cs.iit.edu/~scs/assets/files/Enosis.pdf) and *Syndesis: Mapping Objects to Files for a Unified Data Access System* (http://www.cs.iit.edu/~scs/assets/files/Syndesis.pdf).

Another important task of Hermes adapters is to collect I/O profiling that supports the detection of I/O patterns.



Hermes Adapters

Contents

STDIO Adapter

The STDIO adapter assumes a paged view of a logical file, a byte sequence. The paged view uses a fixed page size (e.g., 1 MB) with which the logical file is partitioned within the adapter. The STDIO adapter will hint the Hermes library to use the page size as the buffer size configuration within the system. There are three types of APIs within STDIO.

```
- Metadata APIs
  - FILE access APIs
    - fclose
    - fopen
    - fopen64
    - fdopen
    - freopen
    - freopen64
  -  FILE position APIs
    - fseek
    - fseeko
    - fseeko64
    - fsetpos
    - fsetpos64
    - rewind
- Bulk Data APIs
  - fread
  - fwrite
  - fflush
  - fgets
  - fputs
- Byte Data APIs
  - fgetc
  - putw
  - getc
  - getw
  - putc
```

The data structure that can maintain data and metadata info is local to the process. This is because `FILE` structure is in the scope of the process and needs locks using `flockfile`. Additionally, STDIO APIs do not support **shared file access** across multiple processes. Hence, we need to map user's `FILE*` to our internal stat structure. The description of the structure is

```
struct FileID {
  dev_t dev_id_;    /* ID of device containing file */
  ino_t inode_num_; /* inode number */
};
```

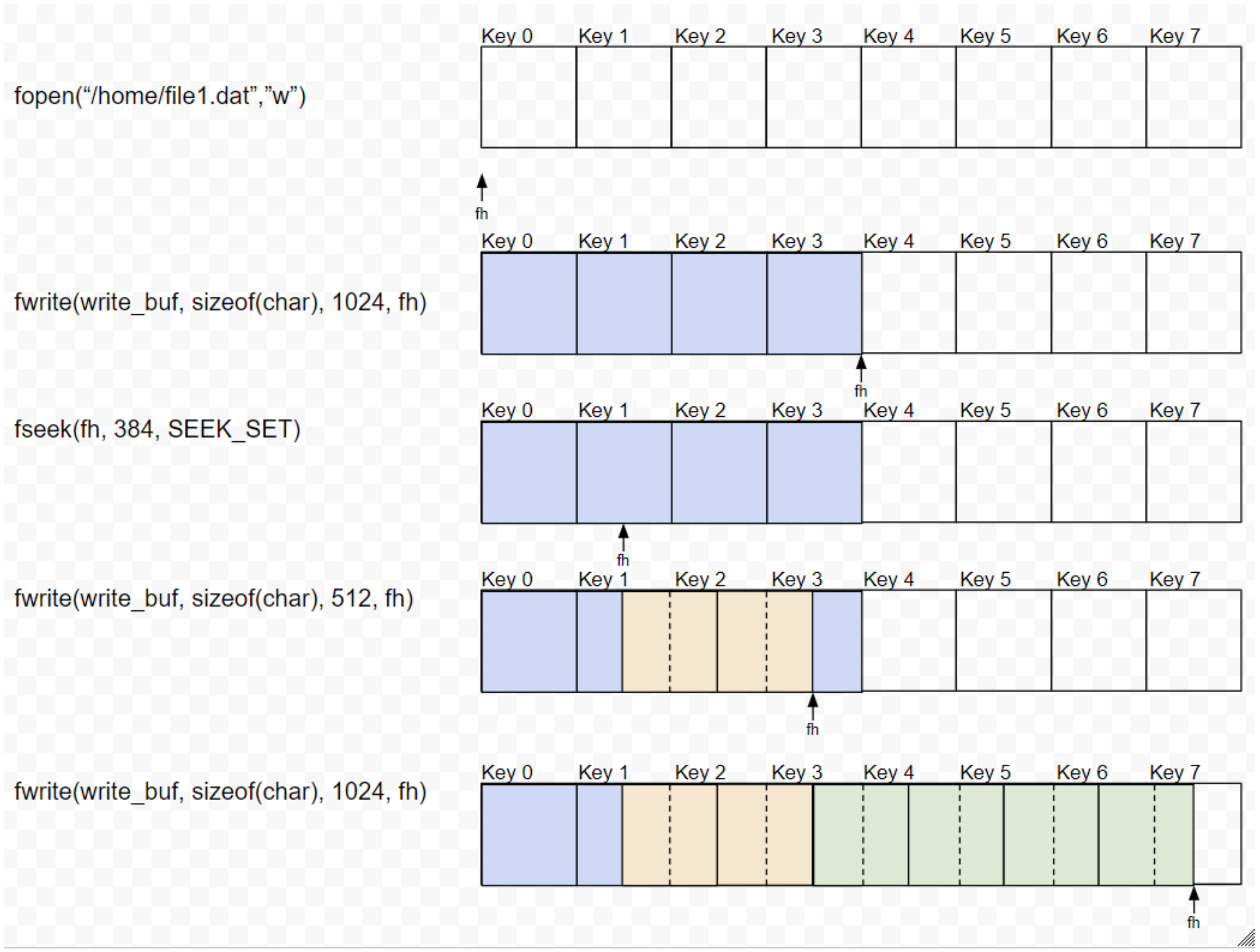
```
struct AdapterStat {
  BucketID     st_bkid;    /* bucket associated with the file */
  i32          ref_count;  /* # of time process opens a file */
  mode_t       st_mode;    /* protection */
  uid_t        st_uid;     /* user ID of owner */
  gid_t        st_gid;     /* group ID of owner */
  off_t        st_size;    /* total size, in bytes */
  off_t        st_ptr;     /* Current ptr of FILE */
  blksize_t    st_blksize; /* blocksize for blob within bucket */
  time_t       st_atime;   /* time of last access */
  time_t       st_mtime;   /* time of last modification */
  time_t       st_ctime;   /* time of last status change */
};
```

The proposed mapping of STDIO to Hermes constructs is as follows

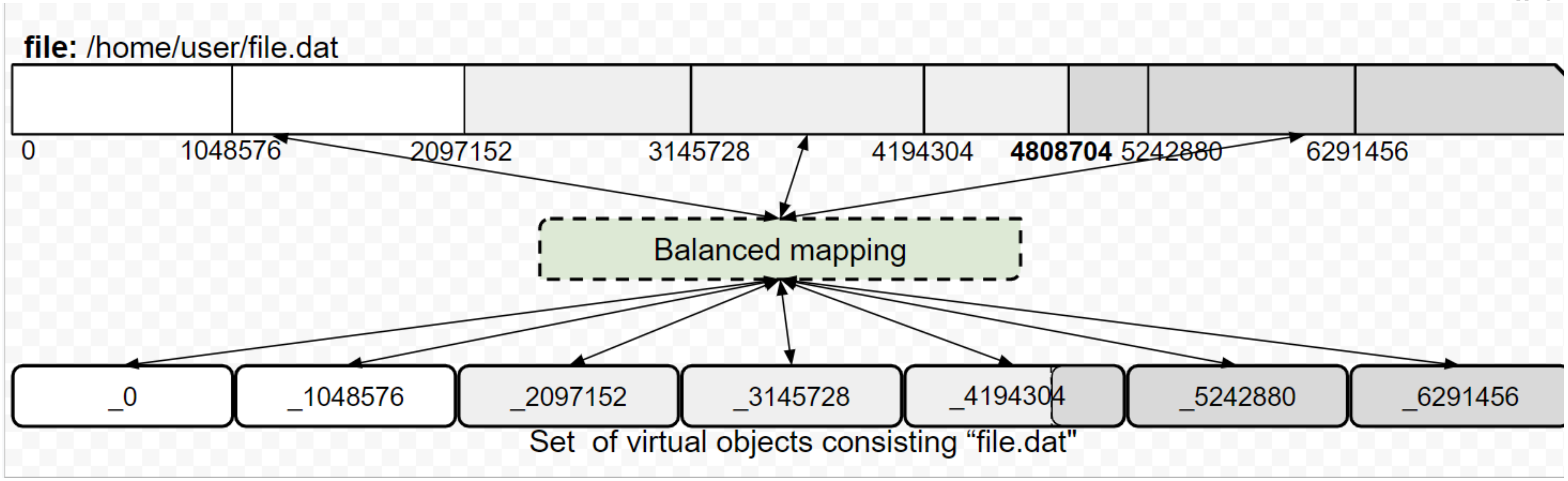
```
FILE -> Bucket
FILE Pages -> Blobs
```

Where each file is a bucket (linked through bucket id). Each File contains pages where each page is referred through a key (corresponds to blob id). An illustration of the idea is presented below.

Hermes File to Object Mapping


fopen(“/home/file1.dat”,”w”)
fwrite(write_buf, sizeof(char), 1024, fh)
fseek(fh, 384, SEEK_SET)
fwrite(write_buf, sizeof(char), 512, fh)
fwrite(write_buf, sizeof(char), 1024, fh)
Key 0
Key 1
Key 2
Key 3
Key 4
Key 5
Key 6
Key 7
fh

Hermes Balanced File Mapping



Mapping of Operations to Hermes structure is as follows

```
- On Open: We create a private bucket with the same name as the filename and an in-memory FILE object. We map this FILE* to AdapterStat.
- On fwrite, fputc, fputs, putc, putw::
  - Check FILE access permisions.
  - We map the current offset and size of fwrite to pages of the logical file using BalanceMappingAlgorithm.
  - The pages correspond to BlobID in Hermes. We write the data into the respective blobs.
    - if BlobID exists we will read the blob, update its contents and write the blob again.
  - Update AdapterStat.
- On fread, fgetc, fgets, getc, getw:
  - Check FILE access permisions.
  - We map the current offset and size of fread to pages of the logical file using BalanceMappingAlgorithm.
  - We read the blobs from Hermes.
  - Update AdapterStat.
  - return the buffer.
- On fflush
  - Flush the bucket contents into a file in PFS.
- On freopen
  - Remap FILE* to new FILE* with new modes. This just involves copying AdapterStat ds to new FILE*.
- On fdopen
  - Remap POSIX fd to FILE* with new modes. This just involves copying AdapterStat ds to new FILE*.
- On fseek, fseeko, fsetpos
  - Change AdapterStat.st_ptr to match the ptr over different SEEK modes.
- On rewind
  - Change AdapterStat.st_ptr to 0.
```

Metadata structures

The structure needed to build and maintain the above semantics is

```
std::unordered_map<FileID, AdapterStat> stat;
```

Multi Thread support

In most cases, we assume we don't need multi-threading support as we will be employing process-level parallelism using MPI. However, if we are supporting Multi-Threading in the adapter, each critical path would be locked using `flockfile` on the in-memory file.

Questions/Considerations

```
- STDIO does not support inter-process file sharing (only POSIX does).
- The Byte data APIs described above, perform I/O in byte granularity. In such a case the adapter performs blob updates in bytes.
  - We need to either aggregate these requests in Adapter (?) or
  - Optimize this special case within Hermes through hints.
```

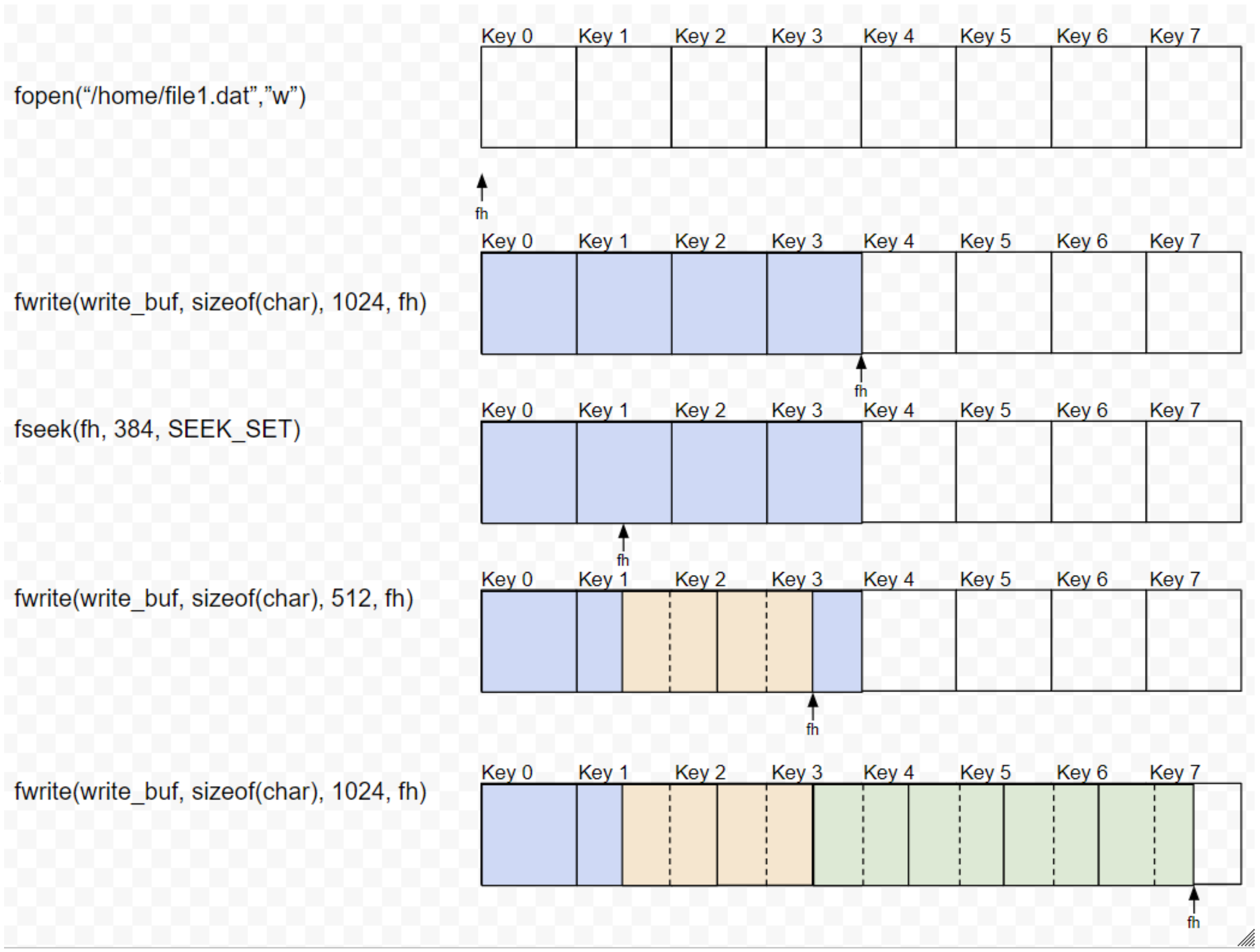
POSIX Adapter

Unix Adapter

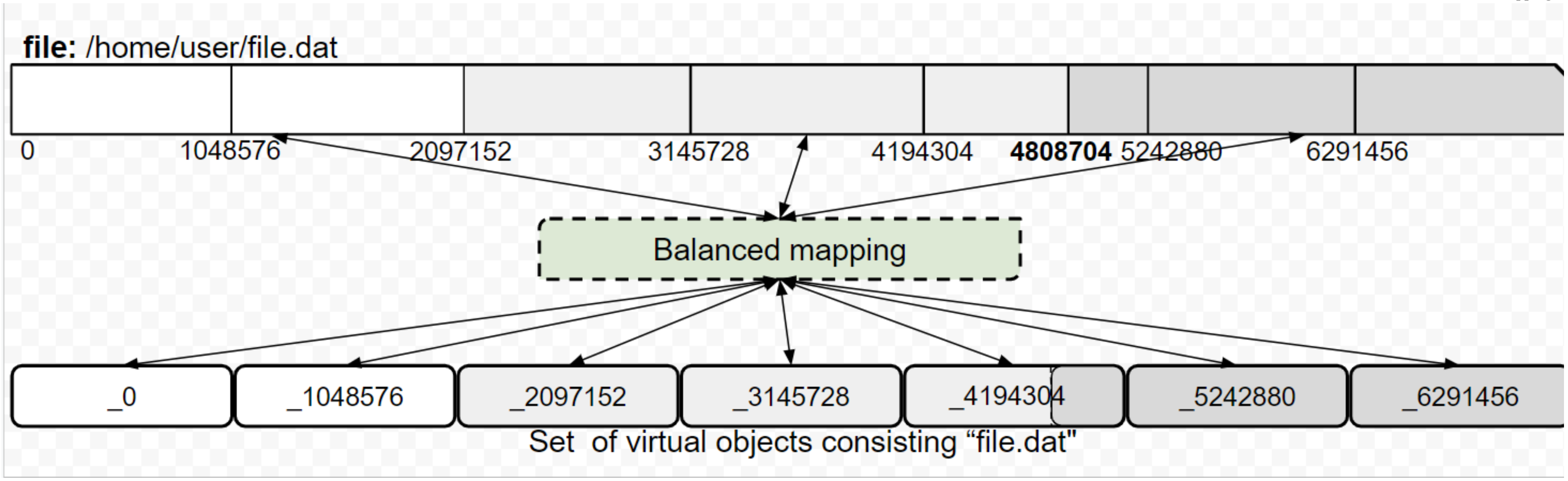
The Unix adapter assumes a paged view of a logical file, a byte vector.

What's the relationship between the page size and the pre-defined buffer sizes in buffer pools?

Hermes File to Object Mapping


fopen("/home/file1.dat","w")
fwrite(write_buf, sizeof(char), 1024, fh)
fseek(fh, 384, SEEK_SET)
fwrite(write_buf, sizeof(char), 512, fh)
fwrite(write_buf, sizeof(char), 1024, fh)
Key 0
Key 1
Key 2
Key 3
Key 4
Key 5
Key 6
Key 7
fh

Hermes Balanced File Mapping



This adapter supports a subset of [[1] (https://pubs.opengroup.org/onlinepubs/9699919799/)[POSIX.1]] calls (see below) related to file I/O. It also supports /poor-man's parallel I/O/ to multiple independent files (MIF).

Metadata structures

We call them tables, but they are maps

File table (FT)

The **File ID** is obtained from `fstat`. The first two fields of the returned `struct stat` uniquely identify a file.

```
File ID := (st_dev, st_ino)
```

```
File ID -> (Bucket ID, Page Size, Reference Count)
```

Do we need to track EOA/EOF?

File descriptor table (FDT)

```
File Desc. -> (File ID, Cursor, Append Mode)
```

Working set (WS)

The set of page keys currently present in file buckets.

```
File Page Key := (File ID, Page Key)
```

Working set statistics (WSS)

```
File Page Key -> (read count, write count)
```

WARNING: Generally, the key sets of WS and WSS will be different. The former is a subset of the latter.

Supported APIs

open

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

```
- Open/create the file, obtain new file descriptor
- Compute File ID
- IF File ID in FT:
  - Increment Reference Count
- ELSE:
  - Call flock and place an exclusive lock on the file
  - Acquire a Hermes bucket
  - Create a new entry in FT
  - Release the lock
- Add entry to FDT
- RETURN file descriptor
```

write

```
ssize_t write(int fd, const void *buf, size_t count);
```

```
- Compute File Page Key Range
- IF File Desc. IN Append Mode:
  - Move Cursor to end of file
- FORALL File Page Key IN File Page Key Range:
  - IF writing full page:
    - Call hermes::put on file bucket w/ File Page Key and user buffer
  - ELSE:
    - IF File Page Key in WS:
      - Call hermes::get to retrieve buffer associated with File Page Key
    - ELSE:
      - Call client_read to retrieve the data
    - Merge w/ user buffer
    - Call hermes::put
  - UNLESS File Page Key in WSS:
    - Add (File Page Key, (0,0)) to WSS
  - Increment write count for File Page Key
- Advance Cursor for this File Desc.
- RETURN the number of bytes written
```

read

```
ssize_t read(int fd, void *buf, size_t count);
```

```
- Compute File Page Key Range
- FORALL File Page Key in Page Key Range:
  - IF File Page Key in WS:
    - hermes::get the data
  - ELSE:
    - Call client_read to retrieve the data
    - hermes::put the data (optional?)
  - IF reading full page:
    - Copy data to user buffer
  - ELSE:
    - Merge data into user buffer
```

```
      - UNLESS File Page Key in WSS:
        - Add (File Page Key, (0,0)) to WSS
      - Increment read count for File Page Key
    - Advance Cursor for this File Desc.
    - RETURN the number of bytes read
```

lseek

```
off_t lseek(int fd, off_t offset, int whence);
```

```
    - Call lseek
    - Update the Cursor for File Desc. fd
    - RETURN the new offset
```

posix_fallocate

```
     int posix_fallocate(int fd, off_t offset, off_t len);
```

close

```
     int close(int fd);
```

```
    - Call client_close
    - Remove the corresponding entry from FDT
    - Decrement the Reference Count
    - IF Reference Count == 0:
      - Call client_flock and place an exclusive lock on the file
      - Release the Hermes bucket
      - Remove the entry from FT
      - Release the lock
```

System calls in H5FDsec2.c

- open
- close
- read, pread
- write, pwrite
- lseek
- stat
- ftruncate
- flock
- fallocate (?)

References

- GOTCHA is a library for wrapping function calls in shared libraries (https://github.com/LLNL/GOTCHA)