DirectFB Fusion kernel module: introduction and API

Niels Roest

niels@directfb.org

DirectFB Fusion kernel module: introduction and API

by Niels Roest

Table of Contents

Chapter 1. Introduction

Fusion is a Inter-Process-Communication (IPC) kernel module with a high level of abstraction. It offers primitives like skirmishes (locks), calls, properties and shared memory pools. You can query the status of the primitives via an entry in the /proc filesystem.

This chapter aims at giving you an overview about what Fusion is, so that the dry API listing in the following chapters makes sense as a whole.

Fusion is the central spider in the web of IPC. In the Fusion philosophy, processes can 'speak' to each other through Fusion only. They speak to each other by using *Fusion primitives*. Any process can request Fusion to create a primitive instance, which can then be accessed and used by all connected Fusion processes through a simple ID. Refer to the API chapter for explanation of each primitive type in more detail.

The primivites are in short: *call*: to perform (a)synchronous calls to other processes; *ref*: reference counter with watch functionality; *reactor*: asynchronous event distribution with reception confirmation; *skirmish*: counting lock; *property*: multi-level locking; *shmpool*: shared memory usage arbitration.

You can use the primitives with the ioctl() API as described in this document, but within the DirectFB project (www.directfb.org) a wrapper has been written which can be used as an entry point instead. This library is simply called Fusion too. In some cases the wrapper adds extra abstraction, for instance: if you do a synchronous call to another process you need to have a dispatcher thread running in that process; the fusion library creates this thread and offers a callback handler subscription.

Fusion relies for a large part on shared memory to not hamper the speed of data transmission. For instance, with DirectFB (build on top of Fusion) processes are allowed to write to the display buffer themselves, where Fusion takes care of arbitrating access and sharing information.

Fusion has the concept of *World*. A world is the environment in which fusion users operate and communicate; it provides the plaform for clients to discover each other. For example, Fusion primitives are identified and used with a Worldwide ID. Each World has its own filesystem node. Fusion allows multiple worlds to run at the same time, separate from each other. This makes it possible to have multiple Fusion-based protocols running alongside.

A participant in a world is called a *fusionee*. Every user-space application that opens and enters a World will obtain a unique Fusion ID. Note that it is therefore possible, but unusual, for an application to get multiple Fusion IDs. The Fusion ID is used as a handle to target other fusionees. To simplify the implementation of single-acces hardware, every world has at most one fusionee which is also Master, the others being Slave fusionees.

Chapter 2. Internal design

..todo..

2.1. ...

...

Chapter 3. API: Application Programming Interface

This section describes how to use the Fusion kernel module. It mainly lists all ioctls of Fusion.

3.1. Opening the node

All Fusion communication is done via /dev nodes. In the current installment, Fusion uses a "char" node with major 250. By convention, the nodes are called /dev/fusion0, /dev/fusion1, etc. with minor 0,1, etc. for a total of 8 nodes. Each separate node is a separate, independent Fusion World. All applications wishing to IPC with each other shall open the same node.

3.1.1. synopsis

To open a Fusion node:

```
int fd;
fd = open( "/dev/fusion/0", flags );
```

3.1.2. description

The `flags` argument has the following meaning:
`flags  = O_RDWR;` default value.
`flags |= O_NONBLOCK;` add this for a non-blocking read call.
`flags |= O_EXCL;` add this to be the Fusion Master of this World.
`flags |= O_APPEND;` add this to be a Fusion Slave in this World.

Master is the other word for being the first Fusionee. You can use the special open call to make sure you are Master. If you happen to be the first to open a World, you will be its Master as well. If you wish to be a Fusion Slave, but the World is still to be opened, a subsequent ioctl to enter the world will block until the Master has 'unblocked' the World. These ioctls are explained further on. Please note that Fusion does not safeguard the Master role after the open call! The role is only fixed after the World enter ioctl. There are two options to protect the Master role: either unify the open-enter pair in your code, or make sure that all intended Slaves will use the slave open call variant.

The terms Master and Slave have no significance other than startup-behaviour and World-wide initialization calls (so noted where applicable). In DirectFB context, the Master role is generally reserved for the process that controls the graphic hardware: communication to the graphic's card IO-space is

handled with Fusion calls towards the Fusion Master process, which will simplify to regular C-calls if initiated by the Master itself.

3.1.3. return values

`EBUSY` request to be Master, but this world already has a Fusion Master.
`EINTR` an interrupt occurred. No changes were made.
`ENOMEM` insufficient kernel memory.

3.2. Other file operations

If your process has successfully opened the fusion node, the following operations can be performed additionally.

3.2.1. synopsis

```
int close( int fd );
ssize_t read( int fd, void *buf, size_t count );
int ioctl( int fd, int request, ... );
void *mmap( void *start, size_t length, int prot, int flags, int fd, off_t offset );
```

There is no write call. In tandem with the read call, if it is non-blocking, you should use the select call to implement non-busy waiting. The parameters for these functions are all standard and are not further explained.

3.2.2. description

The close call will leave the World and close the node for this Fusionee. If all Fusionees in a World have left, the World will be closed and cleaned-up. A left Fusion Master will not be replaced, not even by a new open call.

The read call will try to read as many complete messages (in order) as will fit in the provided buffer. Messages are always 4-byte aligned (see example below) so you may find stuff bytes between consecutive messages. The return value is the byte size of the returned messages (including stuffing). It is not possible to write messages. All messages are created as a result of ioctl calls. The receivable messages are defined together with the corresponding ioctl description, but follow the following basic structure:

```
typedef struct {
     FusionMessageType   msg_type;
     int                 msg_id;
```

```
     int                 msg_size;
     int                 msg_channel;

     /* message data follows */
} FusionReadMessage;
```

`msg_type` is one of `FMT_CALL`, `FMT_REACTOR`, `FMT_SHMPOOL` or `FMT_SEND`, which means that this message is due to activity from call, reactor or shared memory handling, or as the result of a SEND_MESSAGE ioctl. `msg_id` is the free format message identifier (for FMT_SEND) or the call, reactor or pool ID. `msg_size` is the payload size. `msg_channel` is the optional or reactor channel. The payload data is located immediately after the header.

The following simplified example reads messages:

```
char  buf[BUF_SIZE]
char *buf_p = buf;
int   fd    = open("/dev/fusion/0", O_RDWR | O_NONBLOCK );
int   len   = read( fd, buf, BUF_SIZE );
while (buf_p < buf + len)
{
    FusionReadMessage *header = (FusionReadMessage*) buf_p;
    void              *data   = buf_p + sizeof(FusionReadMessage);
    ...handle message...
    buf_p = data + ((header->msg_size + 3) & ~3);
}
```

The ioctl calls are described in more detail in the following sections. Basically, the Fusion ioctls accept 1 argument, which is normally a pointer to a struct that can hold input and output parameters. For example, if the synopsis is `ioctl( fd, FUSION_ENTER, FusionEnter )` then you can use something like this:

```
#include <fusion.h>
int fd = open("/dev/fusion/0", O_RDWR | O_NONBLOCK );
FusionEnter enter;
...init enter.api...
ret = ioctl( fd, FUSION_ENTER, &enter );
...store fusion_id...
```

With the mmap call a shared area is created. This is a small area of memory, the provided size may not be over a kernel page size (commonly minimally 4KB). The same memory area will be mapped for all Fusionees of the same World, so it is possible to share a World global data. The first mmap call must be done by the Fusion Master, this reserves the kernel memory.

3.2.3. return values

The following can be returned by any of the functions:

`0` on success.

`EINVAL` Input parameter out-of-range.
`EFAULT` Kernel memory fault.
`EINTR` an interrupt occurred. No changes were made.
The following are additional return values of the read call:
`EMSGSIZE` If the first message does not fit in the buffer.
`EAGAIN` For a non-blocking read: no messages available.
The following are aditional return values of the mmap call:
`EPERM` First mmap not performed by Fusion Master.
`ENOMEM` No kernel memory left.

3.3. IOCTL: Global fusion commands

Described are the basic ioctl operations.

3.3.1. synopsis

```
     ioctl( fd, FUSION_ENTER           , FusionEnter       )
     ioctl( fd, FUSION_UNBLOCK                             )
     ioctl( fd, FUSION_KILL            , FusionKill        )
     ioctl( fd, FUSION_ENTRY_SET_INFO  , FusionEntryInfo   )
     ioctl( fd, FUSION_ENTRY_GET_INFO  , FusionEntryInfo   )
     ioctl( fd, FUSION_FORK            , FusionFork        )
     ioctl( fd, FUSION_SEND_MESSAGE    , FusionSendMessage )
```

3.3.2. FUSION_ENTER

```
typedef struct {
     struct {
          int            major;         /* [in] */
          int            minor;         /* [in] */
     } api;

     FusionID            fusion_id;     /* [out] */
} FusionEnter;
```

Enter this Fusion World. You must enter the World after opening the Fusion node. Only then, you will be assigned a unique communication handle: your Fusion ID. For all Slaves, this ioctl will block until the Master has 'unblocked' the World with `FUSION_UNBLOCK`.

`major` and `minor` must be filled with the requested API. Supported are 3.x, 4.x and 8.x, which correspond to DirectFB 1.0, 1.1 and >= 1.2 respectively. This document only describes 8.x, which is recommended for new development. The Master will determine the API. The Slaves must be in the

interval [major.0, major.minor]. The `fusion_id` is returned on successful completion. This is `FUSION_ID_MASTER` for the Master.

Return values (besides 0, `EFAULT` and `EINTR`):
`ENOPROTOOPT` Unsupported API version, or Master has selected a non-compatible API version.

3.3.3. FUSION_UNBLOCK

Unblocks the Fusion World. Must be called by the Fusion Master to allow Slaves in this World.

Return values (besides 0, `EFAULT` and `EINTR`):
`EPERM` Not called by Fusion Master.

3.3.4. FUSION_KILL

```
typedef struct {
     FusionID            fusion_id;
     int                 signal;
     int                 timeout_ms;
} FusionKill;
```

Send a signal to other fusionees, corresponding to the 'kill' program. `fusion_id` is the target fusionee, 0 means all but ourself. The ioctl will return success if there is no such fusionee. `signal` is the signal to be delivered, e.g. `SIGTERM`. `timeout_ms` is the requested time-out in milliseconds, -1 means no timeout, 0 means infinite, otherwise the maximum time to wait until at least one fusionee is terminated. This means that if you send a non-terminating signal with a specified timeout this ioctl will not return.

Return values (besides 0, `EFAULT` and `EINTR`):
`ETIMEDOUT` Timeout expired.

3.3.5. FUSION_ENTRY_SET_INFO and FUSION_ENTRY_GET_INFO

```
typedef struct {
     FusionType          type;
     int                 id;
     char                name[FUSION_ENTRY_INFO_NAME_LENGTH]; /* [in] or [out] */
} FusionEntryInfo;
```

Store or read a human readable name to a Skirmish, Property, Reactor, Ref or Shmpool. This name is only used for these ioctls and in the `/proc` filesystem. `type` is one of `FT_REF`, `FT_SKIRMISH`,

`FT_PROPERTY`, `FT_REACTOR` or `FT_SHMPOOL`, for respectively reference, skirmish, property, reactor and shared memory pool. `id` is the object identifier. `name` is the name of the object.

Return values are `ENOSYS`, `EFAULT` or `EINTR` for failure, 0 for success.

3.3.6. FUSION_FORK

```
typedef struct {
 FusionID fusion_id;
} FusionFork;
```

Perform a 'fork' of shared memory pools, reactors and local references. This will copy these entities from fusiuonee `fusion_id` to the calling fusionee. Upon return, `fusion_id` holds the calling fusionee ID.

Return values are `EFAULT` or `EINTR` for failure, 0 for success.

3.3.7. FUSION_SEND_MESSAGE

```
typedef struct {
     FusionID            fusion_id;
     int                 msg_id;
     int                 msg_channel;
     int                 msg_size;
     const void         *msg_data;
} FusionSendMessage;
```

Send a Fusion message. The receiver will call `read()` to receive the message. A fusion message is simply a packet of data which is send from one fusionee (the one calling this ioctl) to another (the one calling `read()`. The sender specifies the receiver by fusion ID, the receiver can extract the sender ID from the received mesage. Check the description of `read()` for additional info.

`fusion_id` is the ID of the fusionee to receive the message. `msg_id` is a free-format message identifier. `msg_channel` is an optional channel number. `msg_size` is the message payload size, which lies in the range [0,65536] inclusive. `msg_data` points to the payload data. must not be NULL.

Return values (besides 0, `EFAULT` and `EINTR`):
`EINVAL` msg_size too small (below 0).
`EMSGSIZE` msg_size too big (above 65536).
`ENOMEM` out of kernel memory. Try a smaller msg_size.

3.4. IOCTL: CALL primitive

Describes the CALL primitive related ioctls. Use this to make a syncronous call to another fusionee. A Fusion call can be compared to an RPC (Remote Procedure Call). First, the receiving fusionee creates the call. Then, any fusionee can 'execute' it - this will block the calling thread until the receiver has 'returned' it. The receiver has to poll/select the `read()` function to wait for an incoming call. Since this is hardly how a proper call would work, the Fusion user-space library has abstraction code that allows the receiver to register a handler function, so the whole 'execute' - 'return' path is hidden and the primitive acts like a call proper.

3.4.1. synopsis

```
     ioctl( fd, FUSION_CALL_NEW        , FusionCallNew     )
     ioctl( fd, FUSION_CALL_EXECUTE    , FusionCallExecute )
     ioctl( fd, FUSION_CALL_RETURN     , FusionCallReturn  )
     ioctl( fd, FUSION_CALL_DESTROY    , int               )
```

3.4.2. FUSION_CALL_NEW

```
typedef struct {
     int                 call_id;       /* [out] */
     void               *handler;       /* [in]  */
     void               *ctx;           /* [in]  */
} FusionCallNew;
```

Create a new Fusion call primitive. This will return a unique call ID. A user of the call primitive can use this call ID to perform (or 'execute') a call with some arguments. Fusion will switch context, if needed, to the creator of the call; it will then receive a fusion message, perform the call and send back the return value with FUSION_CALL_RETURN. With this ioctl you specify a call handler (a.k.a. callback function) and a call context. Both are opaque to Fusion, and are simply passed back when a call is executed.

Fusion CALLs are synchronous or, optionally, one-way only. If synchronous, the execute will block until the call has been completed with FUSION_CALL_RETURN. If one-way, the execute will return immediately, and a FUSION_CALL_RETURN should not be send.

`call_id` is the returned unique call ID. `handler` and `ctx` are opaque handles, which are intended for the callback function. This will be passed in every message that results from an execute.

Return values (besides 0, `EFAULT` and `EINTR`):
`ENOMEM` Out of kernel memory.

3.4.3. FUSION_CALL_EXECUTE

```
typedef struct {
     int                 ret_val;       /* [out] */
     int                 call_id;       /* [in]  */
     int                 call_arg;      /* [in]  */
     void               *call_ptr;      /* [in]  */
     FusionCallExecFlags flags;         /* [in]  */
} FusionCallExecute;
```

Execute a call on an existing Fusion call primitive. This will send a Fusion message to the owner/creator of the call ID. This means that the receiving end has the responsability to listen to this message, and to call FUSION_CALL_RETURN.

In `ret_val` the return value is returned. `call_id` is the target call ID. `call_arg` and `call_ptr` are optional arguments that will be passed to the receiver inside the message. `flags` is `FCEF_NONE` for a synchronous or `FCEF_ALL` for an asynchronous call.

The Fusion message that is send consists of the following two structures, send back-to-back. The total message size is thus `sizeof(FusionReadMessage) + sizeof(FusionCallMessage)` rounded up for 4-byte alignment.

```
typedef struct {
     FusionMessageType   msg_type;
     int                 msg_id;
     int                 msg_size;
     int                 msg_channel;
} FusionReadMessage;
typedef struct {
    void               *handler;
    void               *ctx;
    int                 caller;
    int                 call_arg;
    void               *call_ptr;
    unsigned int        serial;
} FusionCallMessage;
```

The first 4 elements have already been described by `read()`, but notice that `msg_type` is FMT_CALL. `handler` and `ctx` are the parameters that were provided with FUSION_CALL_NEW. `caller` is the fusion ID of the caller. Note that Fusion internally also generates calls (e.g. for reactors), in which case this is 0. `call_arg` and `call_ptr` are provided with FUSION_CALL_EXECUTE. `serial` is the call serial which is to be used for FUSION_CALL_RETURN.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`ENOMEM` Out of kernel memory.

3.4.4. FUSION_CALL_RETURN

```
typedef struct {
     int                call_id;
     int                val;
     unsigned int       serial;
} FusionCallReturn;
```

Return the return value to the caller. This will unblock the caller. Only to be called on a synchronous call.

`call_id` is the call ID you want to return. `val` is the return value. `serial`, as received from FUSION_CALL_EXECUTE.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EOPNOTSUPP` Trying to return a one-way call.
`EIDRM` Caller was already unblocked due to a signal. Return value lost.
`ENOMSG` No waiter found. Can point to halted waiting fusionee.

3.4.5. FUSION_CALL_DESTROY

Destroys this call primitive. This will destroy the call, and unlock any pending execute. This can be done only by the same fusionee that created the call primitive. The passed `int` is the call ID.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` We didn't create the call.

3.5. IOCTL: REF primitive

Describes the reference primitive related ioctls. A reference simply implements reference counting. Each instance has a global count and for each fusionee a local count. These counts can be adjusted, tested and watched. Counts must be 0 or higher.

3.5.1. synopsis

```
     ioctl( fd,  FUSION_REF_NEW         , int )
     ioctl( fd,  FUSION_REF_UP          , int )
     ioctl( fd,  FUSION_REF_UP_GLOBAL   , int )
     ioctl( fd,  FUSION_REF_DOWN        , int )
     ioctl( fd,  FUSION_REF_DOWN_GLOBAL , int )
     ioctl( fd,  FUSION_REF_ZERO_LOCK   , int )
     ioctl( fd,  FUSION_REF_ZERO_TRYLOCK, int )
     ioctl( fd,  FUSION_REF_UNLOCK      , int )
```

```
ioctl( fd,  FUSION_REF_STAT        , int )
ioctl( fd,  FUSION_REF_WATCH       , FusionRefWatch )
ioctl( fd,  FUSION_REF_INHERIT     , FusionRefInherit )
ioctl( fd,  FUSION_REF_DESTROY     , int )
```

3.5.2. FUSION_REF_NEW

Create a new reference primitive. The `int` contains the created reference ID. The reference's global and local counts are set to 0. Return values are `EFAULT` or `EINTR` for failure, 0 for success.

3.5.3. FUSION_REF_UP_GLOBAL and FUSION_REF_DOWN_GLOBAL

Increase or decrease the global count of this reference by 1. The `int` contains the reference ID. The reference shall not be in a locked state.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EAGAIN` Reference is locked.
`EIO` Count is 0, cannot be decreased further.

3.5.4. FUSION_REF_UP and FUSION_REF_DOWN

Increase or decrease the local count of this reference by 1. A reference has, besides a single global count, for every fusionee a local count, which is only visible that fusionee. The `int` contains the reference ID. The reference shall not be in a locked state.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`ENOMEM` No kernel memory left.
`EAGAIN` Reference is locked.
`EIO` Count is 0, cannot be decreased further.

3.5.5. FUSION_REF_ZERO_LOCK and FUSION_REF_ZERO_TRYLOCK

Lock a reference to zero. The total count (the global count and all local counts) must be zero. On success, the counts are not modifiable anymore. If the total count is not zero yet, LOCK will wait indefinately, while TRYLOCK will return an error code. The `int` contains the reference ID. The reference shall not be in a locked or, for LOCK, watched state.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):

`EIO` You already locked this reference.

`EAGAIN` Reference is locked by another fusionee.

`EACCES` LOCK is tried, but the reference is being watched.

`ETOOMANYREFS` TRYLOCK is tried, but the total count is not 0.

`EIDRM` Internal error.

3.5.6. FUSION_REF_UNLOCK

Unlock the reference. The `int` contains the reference ID.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):

`EIO` Not the lock owner, or not locked.

3.5.7. FUSION_REF_STAT

Return the total count of this reference (the global count and all local counts). The `int` contains the reference ID. Return values are `EINVAL`, `EFAULT` or `EINTR` for failure, 0 for success.

3.5.8. FUSION_REF_WATCH

```
typedef struct {
     int                 id;
     int                 call_id;
     int                 call_arg;
} FusionRefWatch;
```

Watch a reference. If the total count drops to 0, a call will be executed.

`id` contains the reference ID. `call_id` is the call ID that will be executed. `call_arg` is an optional parameter that will be part of the call message. The call message is scheduled below:

```
typedef struct {
     FusionMessageType   msg_type;
     int                 msg_id;
     int                 msg_size;
     int                 msg_channel;
} FusionReadMessage;
typedef struct {
    void               *handler;
    void               *ctx;
    int                 caller;
    int                 call_arg;
```

```
    void               *call_ptr;
    unsigned int        serial;
} FusionCallMessage;
```

`caller` is 0. `call_arg` is provided by this ioctl. `call_ptr` is set to NULL. `serial` is not used. The call is send asynchronously, so do not send a FUSION_CALL_RETURN. For an explanation of the other fields inside the call message, check FUSION_CALL_EXECUTE.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EACCES` Current process ID is not the creating process ID.
`EIO` The total count is already 0. No watch installed.
`EBUSY` Reference is already being watched.

3.5.9. FUSION_REF_INHERIT

```
typedef struct {
     int                 id;
     int                 from;
} FusionRefInherit;
```

Inherit the local count from another reference. `id` is the own ID, `from` is the ID to inherit from.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EBUSY` Reference is already inherited.

3.5.10. FUSION_REF_UNLOCK

Unlock the reference. The `int` contains the reference ID.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` Not the lock owner, or not locked.

3.6. IOCTL: SKIRMISH primitive

Describes the skirmish primitive related ioctls. A skirmish can be seen as a counting lock.

3.6.1. synopsis

```
ioctl( fd, FUSION_SKIRMISH_NEW        , int )
ioctl( fd, FUSION_SKIRMISH_PREVAIL    , int )
```

```
ioctl( fd, FUSION_SKIRMISH_SWOOP      , int )
ioctl( fd, FUSION_SKIRMISH_DISMISS    , int )
ioctl( fd, FUSION_SKIRMISH_DESTROY    , int )
ioctl( fd, FUSION_SKIRMISH_LOCK_COUNT , int[2] )
ioctl( fd, FUSION_SKIRMISH_WAIT       , FusionSkirmishWait )
ioctl( fd, FUSION_SKIRMISH_NOTIFY     , int )
```

3.6.2. FUSION_SKIRMISH_NEW

Create a new skirmish primitive. The `int` contains the created skirmish ID. Return values are `EFAULT` or `EINTR` for failure, 0 for success.

3.6.3. FUSION_SKIRMISH_PREVAIL and FUSION_SKIRMISH_SWOOP

Tries to take the skirmish. The `int` contains the reference ID. If the skirmish is already taken, PREVAIL will wait forever until it is dismissed, SWOOP will always return directly with an error. If the call is successful, the skirmish count will be increased by 1.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EAGAIN` returned by SWOOP: reference is locked by another fusionee.

3.6.4. FUSION_SKIRMISH_DISMISS

Releases the skirmish. The `int` contains the reference ID. PREVAIL will wait forever, SWOOP will always return directly. If the call is successful, the skirmish count will be decreased by 1. If the count is 0, a skirmish notification is send.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` The skirmish is not taken, or not taken by this fusionee.

3.6.5. FUSION_SKIRMISH_DESTROY

Destroys a skirmish primitive. The `int` contains the skirmish ID. Return values are `EINVAL`, `EFAULT` or `EINTR` for failure, 0 for success.

3.6.6. FUSION_SKIRMISH_LOCK_COUNT

Returns the skirmish count. `int[0]` contains the skirmish ID. On output, `int[1]` contains the skirmish count for this fusionee only. This means that if you were not the fusionee that did PREVAIL or SWOOP, the count will be 0. Return values are `EINVAL`, `EFAULT` or `EINTR` for failure, 0 for success.

3.6.7. FUSION_SKIRMISH_WAIT

```
typedef struct {
     int                 id;
     unsigned int        timeout;
     unsigned int        lock_count;
     unsigned int        notify_count;
} FusionSkirmishWait;
```

Wait for somebody else to take, and subsequently release, the skirmish. You must first take the skirmish yourself with either PREVAIL or SWOOP. Then you can call WAIT. This will release the skimish, and wait for another fusionee to take it, NOTIFY, and release it. Then the skirmish will be taken again. It is possible that the skirmish is taken and released more than once during one WAIT.

`id` contains the skirmish ID. `timeout` is a timeout in milliseconds. 0 means forever. `lock_count` and `notify_count` are required for interrupt handling. You must initialize both with 0. When the WAIT returns with `EINTR`, you must not touch these fields, and simply call the ioctl again.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` You did not take this skirmish.

3.6.8. FUSION_SKIRMISH_NOTIFY

Notify waiting skirmishes. As soon as you release the skirmish, any fusionees having performed a WAIT will be allowed to continue. The int `id` is the skirmish ID. Return values are `EINVAL`, `EFAULT` or `EINTR` for failure, 0 for success.

3.7. IOCTL: PROPERTY primitive

Describes the property primitive related ioctls.

3.7.1. synopsis

```
     ioctl( fd, FUSION_PROPERTY_NEW      , int)
```

```
ioctl( fd, FUSION_PROPERTY_LEASE    , int)
ioctl( fd, FUSION_PROPERTY_PURCHASE , int)
ioctl( fd, FUSION_PROPERTY_CEDE     , int)
ioctl( fd, FUSION_PROPERTY_HOLDUP   , int)
ioctl( fd, FUSION_PROPERTY_DESTROY  , int)
```

3.8. IOCTL: REACTOR primitive

Describes the reactor primitive related ioctls. A reactor is used to distribute events. If a fusionee uses the DISPATCH ioctl, all attached fusionees will receive a call message. It is possible to fine-grain the target audience by using channels: each reactor supports 1024 channels, and each channel can be independantly attached, detached, or dispatched to.

3.8.1. synopsis

```
ioctl( fd, FUSION_REACTOR_NEW                   , int)
ioctl( fd, FUSION_REACTOR_ATTACH                , FusionReactorAttach)
ioctl( fd, FUSION_REACTOR_DETACH                , FusionReactorDetach)
ioctl( fd, FUSION_REACTOR_DISPATCH              , FusionReactorDispatch)
ioctl( fd, FUSION_REACTOR_DESTROY               , int)
ioctl( fd, FUSION_REACTOR_SET_DISPATCH_CALLBACK , FusionReactorSetCallback)
```

3.8.2. FUSION_REACTOR_NEW

Create a new reactor primitive. The `int` contains the created reactor ID. Return values are `EFAULT` or `EINTR` for failure, 0 for success.

3.8.3. FUSION_REACTOR_ATTACH and FUSION_REACTOR_DETACH

```
typedef struct {
     int                 reactor_id;
     int                 channel;
} FusionReactorAttach;

typedef struct {
     int                 reactor_id;
     int                 channel;
} FusionReactorDetach;
```

Attach or detach a fusionee to a reactor. Both are counting, so the amount of attaching must match the amount of detaching to be completely detached. `reactor_id` is the reactor ID, `channel` is the selected reactor channel number. This number can range from 0 to 1023, inclusive. Note that the current implementation uses less memory if you avoid high channel numbers.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` for DETACH: not attached.
`ENOMEM` No more kernel memory.
`EIDRM` Reactor destruction is pending.

3.8.4. FUSION_REACTOR_DISPATCH

```
typedef struct {
     int                 reactor_id;
     int                 channel;
     int                 self;
     int                 msg_size;
     const void         *msg_data;
} FusionReactorDispatch;
```

Dispatch a message in a reactor. All attached fusionees will receive this message, and optionally the sender as well, if attached. `reactor_id` is the reactor ID. `channel` is channel number. `self` (boolean) 0 means the sending fusionee will not receive the message. `msg_size` is the size of the message data. `msg_data` is a pointer to the message data. The message data will be copied for each attached fusionee. Note that in the current implementation, when copying the message data, an out-of-kernel-memory error will be discarded (not reported). The message header has the following layout (check `read()` for more info):

```
typedef struct {
     FusionMessageType   msg_type;
     int                 msg_id;
     int                 msg_size;
     int                 msg_channel;
} FusionReadMessage;
```

`msg_type` is FMT_REACTOR. `msg_id` is the reactor ID. `msg_size` is the total message size, inclusive payload. `msg_channel` is the channel number. The message data itself will be send directly after this header.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`ENOMEM` Out of kernel memory.
`EIDRM` Reactor is pending destruction.
`EMSGSIZE` Message size exceeds 65536 bytes.

3.8.5. FUSION_REACTOR_DESTROY

Destroys the reactor. If there are still fusionees attached, the destruction will be delayed until the last fusionee detaches itself. The provided `int` is the reactor ID. Return values are `EINVAL`, `EFAULT` or `EINTR` for failure, 0 for success.

3.8.6. FUSION_REACTOR_SET_DISPATCH_CALLBACK

```
typedef struct {
     int                 reactor_id;
     int                 call_id;
     int                *call_ptr;
} FusionReactorSetCallback;
```

Install a callback handler on dispatch. This will execute a single call after a dispatch has been performed on this reactor and all attached fusionees have read the messages (via `read()`). The call will always execute, also when nobody has attached to the reactor's channel of the dispatch. `reactor_id` is the reactor ID to watch. `call_id` is the call ID to send the call to. `call_ptr` is an optional parameter that will be passed back in the call. The call execute will result in the following message:

```
typedef struct {
     FusionMessageType   msg_type;
     int                 msg_id;
     int                 msg_size;
     int                 msg_channel;
} FusionReadMessage;
typedef struct {
    void               *handler;
    void               *ctx;
    int                 caller;
    int                 call_arg;
    void               *call_ptr;
    unsigned int        serial;
} FusionCallMessage;
```

The first 4 elements have already been described by `read()`, but notice that `msg_type` is FMT_CALL. `handler` and `ctx` are the parameters that were provided with FUSION_CALL_NEW. `caller` is 0. `call_arg` is the reactor's channel number. `call_ptr` is the `call_ptr` provided above. `serial` is 0. The call is asynchronous so you must not do a CALL_RETURN.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIDRM` Reactor is pending destruction.

3.9. IOCTL: SHMPOOL primitive

Describes the shared memory pool primitive related ioctls. Fusion provides the means to share memory areas, but does not implement the necessary mapping. Conceptually, Fusion controls a single contiguous memory area, and each request for a pool will reserve a chunk from this area. In principle, the location of this area in real memory is up to the application. An easy implementation will be to follow the address suggestions done by Fusion and perform a fixed mapping (mmap with MAP_FIXED); the suggested addresses are normally located in non-occupied virtual memory space. This memory mapping abstraction, as well as an implementation for memory allocation handling (malloc family) is already included in the user-space Fusion library.

3.9.1. synopsis

```
     ioctl( fd, FUSION_SHMPOOL_NEW      , FusionSHMPoolNew)
     ioctl( fd, FUSION_SHMPOOL_ATTACH   , FusionSHMPoolAttach)
     ioctl( fd, FUSION_SHMPOOL_DETACH   , int)
     ioctl( fd, FUSION_SHMPOOL_DISPATCH , FusionSHMPoolDispatch)
     ioctl( fd, FUSION_SHMPOOL_DESTROY  , int)
```

3.9.2. FUSION_SHMPOOL_NEW

```
typedef struct {
     int                 max_size;      /* [in]  */
     int                 pool_id;       /* [out] */
     void               *addr_base;     /* [out] */
} FusionSHMPoolNew;
```

Create a new shared memory pool primitive. This will reserve an area of memory, and link it to the returned pool ID. After creating the pool, you still have to ATTACH to it, and possibly DISPATCH it, since the size after creation is always 0. The client is responsible for making sure that the returned memory address and will be mapped in the application space.

`max_size` is the maximum pool size. The current size after creation is always 0. `pool_id` is the pool ID of the created pool. `addr_base` is the suggested memory location of the pool. This is a page aligned address.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`ENOSPC` Out of virtual memory.

3.9.3. FUSION_SHMPOOL_ATTACH

```
typedef struct {
     int                 pool_id;       /* [in]  */
```

```
     void               *addr_base;     /* [out] */
     int                 size;          /* [out] */
} FusionSHMPoolAttach;
```

Attach this fusionee to a shared memory pool. This will increase a counting reference of this fusionee to this pool. You need this to inform yourself about size changes, see DISPATCH. `pool_id` is the pool ID. `addr_base` is the suggested memory location of the pool. `size` is the current size.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`ENOMEM` Out of kernel memory.

3.9.4. FUSION_SHMPOOL_DETACH

Detach the calling fusionee from the pool. Since it is a counting reference, you need to call DISPATCH an equal amount of times as ATTACH. The provided `int` is the pool ID.

Return values (besides 0, `EINVAL`, `EFAULT` and `EINTR`):
`EIO` You did not attach to this pool.

3.9.5. FUSION_SHMPOOL_DISPATCH

```
typedef struct {
     int                 pool_id;
     int                 size;
} FusionSHMPoolDispatch;
```

Change the size of the shared memory pool. `pool_id` is the pool ID. `size` is the new requested size. DISPATCH will send a message to all attached fusionees (except the DISPATCH caller) to inform them about the new size. The message has te following layout (check `read()` for more info):

```
typedef struct {
     FusionMessageType        msg_type;
     int                      msg_id;
     int                      msg_size;
     int                      msg_channel;
} FusionReadMessage;
typedef struct {
     FusionSHMPoolMessageType type;
     int                      size;
} FusionSHMPoolMessage;
```

`msg_type` is FMT_SHMPOOL. `msg_id` is the pool ID. `msg_size` is the size of the message. `msg_channel` is 0. `type` is FSMT_REMAP. `size` is the new size of the pool. The message is send asynchronously and must not get a reply.

Return values are `EINVAL`, `EFAULT` and `EINTR`) for failure and 0 for success.

3.9.6. FUSION_SHMPOOL_DESTROY

Destroy the pool. The provided `int` is the pool ID. Return values are `EINVAL`, `EFAULT` and `EINTR`) for failure, and 0 for success.

Chapter 4. Information from /proc/fusion

If your application has opened a Fusion node, Fusion will populate the /proc tree with information files. If you opened `/dev/fusion0`, the corresponding files can be found in `/proc/fusion/0/`. These files will be removed when the last file handler to `/dev/fusion0` is closed.

You can simply cat all these files, it will always contain the current state of Fusion. This can be very practical to see for instance the reason of a process being blocked in a Fusion primitive. The files follow a simple convention: each info element occupies a single line, so e.g. `cat reactors | wc -l` will list the number of reactors. The entries are generally listed in the order they were last accessed, so last executed = shown first.

4.1. fusionees

The file `fusionees` shows information about all the connected processes. Each process with an opened filehandle is called a *fusionee*. The first fusionee is sometimes called the Master. This abstraction is carried by the Fusion library, but is not further important in the kernel context. The following lists the output:

```
/proc/fusion/0$ cat fusionees
( 8635) 0x00000001 (   0 messages waiting,      20 received,       0 sent)
( 8640) 0x00000002 (   0 messages waiting,       0 received,      20 sent)
```

Each line gives information about a fusionee in the following format: `(linux process ID) Fusion ID ( messages waiting, received and send )`. The fusionee with Fusion ID of 0x01 is also known as the Fusion Master. The messages are a total of primitive messages send, such as is done with calls, reactors, shared memory pools or generic fusion messages.

4.2. calls

The file `calls` shows information about all the call primitives. The following shows an example output:

```
/proc/fusion/0$ cat calls
1.2 s  (10352) 0x00000001              (1 calls) idle
1.2 s  (10352) 0x00000004              (5 calls) idle
1.2 s  (10352) 0x00000002              (1 calls) idle
1.3 s  (10352) 0x00000006              (1556 calls) idle
-.-    (10352) 0x00000007              (0 calls) idle
-.-    (10352) 0x00000005              (0 calls) idle
-.-    (10352) 0x00000003              (0 calls) idle
```

Each line gives information about a call in the following format: `last access time (linux process ID) Call ID ( calls performed so far ) call state`. The Call ID is a sequence number to identify each call, counting from 1. The call count is increased when the call has been received by the target fusionee. The state can be `idle` or `executing [ Fusionee ID ]`. It is in the executing state when the call is waiting for the return value from the mentioned (target) fusionee.

4.3. properties

The file `properties` shows information about the instantiated properties. The following shows an example output:

```
/proc/fusion/0$ cat properties
-.-    ( 8635) 0x00000001
```

Each line gives information about a property in the following format: `last access time (linux process ID) Property ID state`. Last access time is -.- if the primitive has not yet been accessed. The Property ID is a sequence number to identify each call, counting from 1. The state is empty if this property is still available, otherwise it is: `leased/purchased by Fusionee ID ( lock PID ) lock count`. For further information check out the properties API.

4.4. reactors

The file `reactors` shows information about the active reactors in the system. The following shows an example output:

```
/proc/fusion/0$ cat reactors
1.1 s  ( 8635) 0x00000002  X11 Input               11x dispatch, 0 nodes
0.5 h  ( 8635) 0x00000005  Surface Pool             2x dispatch, 1 nodes
0.5 h  ( 8635) 0x00000004  Layer Region Pool        0x dispatch, 0 nodes
0.5 h  ( 8635) 0x00000003  Layer Context Pool       0x dispatch, 0 nodes
0.5 h  ( 8635) 0x00000001  Virtual Input            0x dispatch, 0 nodes
```

...

4.5. refs

The file `refs` shows information about the reference primitives. The following shows an example output:

```
/proc/fusion/0$ cat refs
2.6 s  ( 8635) 0x00000003  Layer Context Pool        1  4
2.6 s  ( 8635) 0x00000004  Layer Region Pool         1  3
0.5 h  ( 8635) 0x00000005  Surface 800x600 RGB16     1  3
```

```
0.5 h  ( 8640) 0x00000006  SaWMan Process            0  1
0.5 h  ( 8635) 0x00000001  Arena 'DirectFB/Core'     0  2
0.5 h  ( 8635) 0x00000002  SaWMan Process            0  1
```

...

4.6. shmpools

The file `shmpools` shows information about the Shared Memory Pools currently running in this world. The following shows an example output:

```
/proc/fusion/0$ cat shmpools
0.5 h ( 8635) 0x00000005  SaWMan Pool         0x00005230055c0000 [0x103000] - 0x0, 0x dispa
0.5 h ( 8635) 0x00000004  Surface Memory Pool 0x0000523001550000 [0x4061000] - 0x0, 0x disp
0.5 h ( 8635) 0x00000003  DirectFB Data Pool  0x0000523000530000 [0x1019000] - 0x0, 0x disp
0.5 h ( 8635) 0x00000002  DirectFB Main Pool  0x0000523000120000 [0x407000] - 0x0, 0x dispa
0.5 h ( 8635) 0x00000001  Fusion Main Pool    0x0000523000010000 [0x103000] - 0x0, 0x dispa
```

...

4.7. skirmishs

The file `skirmishs` shows information about the Skirmish primitives currently running in this World. The following shows part of an example output:

```
/proc/fusion/0$ cat skirmishs
726  ms  ( 8635) 0x0000001d  Layer Region
726  ms  ( 8635) 0x0000001e  Surface 800x600 RGB16
  3.1 s  ( 8635) 0x00000016  X11 Input
  3.1 s  ( 8635) 0x0000001c  Layer Context
  3.1 s  ( 8635) 0x0000001a  SaWMan
  3.7 s  ( 8635) 0x00000018  Display Layer 0
  0.5 h  ( 8635) 0x00000010  X11 Shm Images
```

...

4.8. stat

The file `stat` shows a summary of what is happening in this World. Each field shows a total count of events since creation. It also shows the currently selected API version of this World. The following is an example output:

```
/proc/fusion/0$ cat stat
Fusion API:8.0
lease/purchase  cede  attach  detach  dispatch  ref up  ref down  prevail/swoop dismiss
         0         0       1       0        22     149       132       17759      17759
```