

Transcendent Memory Interface Specification

Version 0.0.1 - 081202

*Created by: Dan Magenheimer
(update log at end of document)*

Introduction

Transcendent memory (*tmem*) looks to an operating system (OS) like extended RAM, except that *tmem* “RAM” is dynamically variable in size and can only be accessed through a narrow, well-defined but quirky interface. The quirkiness is not at all arbitrary; it allows pages to be managed efficiently. [Need more here]

A *tmem client* is usually an OS, though other types of clients are possible, such as an enterprise application. A *tmem implementation* is usually provided by a hypervisor and accessed via hypercalls, but other implementations are possible, such as an OS-local page compression utility, or a partition manager in a mainframe-class server.

Transcendent memory is made up of a set of pools. Each pool is made up of a set of objects. And each object contains a set of pages. The combination of a pool id, an object id, and a page id uniquely identify a page of *tmem* data. We call this list of three identifiers a *handle-tuple*, or just *handle*. In an operating system for example, the three parts of a handle-tuple can represent a filesystem, a file within that filesystem, and a page within that file.

When a *tmem* pool is created, it is given certain attributes: it can be private or shared, and it can be persistent or ephemeral. Each combination of these attributes provides a different set of useful functionality and also defines a slightly different set of semantics for the various operations on the pool. For example:

- a private+ephemeral (PE) pool might be used as a “second chance page cache” for storing clean page cache pages
- a private+persistent (PP) pool may be useful to an OS as a pseudo-swap device
- a shared+ephemeral (SE) pool may be useful as a cache for a cluster filesystem
- a shared+persistent (SP) pool might be used for interdomain shared memory

Other pool attributes include the size of the page and a version number.

Once a pool is created, operations are performed on the pool. Pages are copied between the OS and *tmem* and are addressed using a handle-tuple (certain sub-page operations are also supported, including an atomic “exchange” operation, but all such operations operate on a single page). Pages and/or objects may also be entirely removed or *flushed* from the pool. When all operations are completed, a pool is destroyed.

1 **Specification**

2
3 [this part needs more info and some structure/organization]

4
5 Note that “persistence” doesn’t imply a lifetime longer than the underlying tmem is running,
6 i.e. data is never written to permanent storage such as a disk.

7
8 The tmem interface provides versioning and requires that an implementation reject attempts
9 to use a version of the spec newer than the version the implementation provides, but may
10 support the semantics for multiple version numbers. Major version numbers of this spec (i.e.
11 version 1.0 and version 2.0) identify incompatible semantics. Minor version number changes
12 (e.g. version 1.1 and 1.2) represent clarifications of the spec within the same version number;
13 no significant semantic differences are allowed. As a result, only major versions are
14 communicated between client and implementation. [BUT NOTE: Until version 0.1 is
15 published, version 0.0.x may be incompatible with 0.0.y and it is the developers
16 responsibility to avoid client/implementation mismatches!]

17
18 Data sizes: A UUID is always 128 bits, a pool_id is always 32 bits, an object_id is always 64
19 bits and a page_id is always 32 bits. Offset and len are also always 32 bits.

20
21 A tmem client provides raw pageframe numbers, “pfns”, and data is copied to and from those
22 the physical memory represented by those pfns. Since the size of a pageframe number is
23 hardware dependent, an implementation capable of handling multiple clients with different
24 pfn size requirements (i.e. an implementation in a hypervisor) must be flexible in providing
25 support for both 32-bit and 64-bit pfns.

26
27 A tmem implementation provides no serialization guarantees (e.g. to an SMP client) other
28 than as required to protect its own internal data structures. [This may require more thought;
29 perhaps a lock_page, lock_object, and/or lock_pool would be useful?] For example, if
30 different client threads are putting and flushing the same page, the results are indeterminate.

31
32 A tmem implementation may support multiple page sizes, ranging from 4K to 128MB, or
33 may support only one, which may be fixed or determined dynamically (i.e. by the first
34 pagesize used).

35
36 In pseudo-code examples, ABC and XYZ represent different handle-tuples, D1 and D2
37 represent pageframes with different data contents, and E represents an empty pageframe.

39 **TMEM_NEW_POOL(UUID, flags)**

40
41 *Create a new pool for transcendent memory (tmem).*

42
43 In the (32-bit) flags parameter:

- 44 • Bit 0 is set for persistence and cleared for ephemeral. Bit 1 is set for shared and cleared
45 for private.

- Bits 4-7 indicate the pagesize to use, with zero meaning 4K and n meaning $((2^{**n}) * 4K)$, allowing for pagesizes up to 128MB. An implementation may restrict the pagesizes supported and will simply reject an attempt to specify an unsupported pagesize.
- Bits 24-31 provide a specification version number, with higher values representing newer versions of the spec. An implementation must reject an attempt to create a pool that requires the semantics of a newer version number than the implementation provides.
- All other bits are reserved for future use.

UUID is ignored for private pools. For shared pools, after one client first creates the shared pool, subsequent clients share the same pool simply by also “creating” a shared pool and supplying the same UUID. Note that a pool-id for a shared pool is not itself shared; that is, two domains will likely refer to the same shared pool using different locally obtained pool-ids.

Returns: int. A negative return value indicates failure; that the implementation has insufficient memory, or the calling domain has exhausted the implementation-dependent limit of allocated pools, or the pagesize is not supported. A return value ≥ 0 both indicates success and is a “pool id”, which must be provided for all subsequent operations on the created pool..

TMEM_DESTROY_POOL(pool_id)

Destroy a tmem pool and free all data

Pool-id must identify a previously created pool.

For persistent pools, note that this operation destroys any data held in that pool, so should only be used when the pool has been emptied or when the caller is certain any remaining data is invalid. For shared pools, all access to the shared pool by the calling client is revoked; the destruction occurs only when the last sharing client destroys the pool.

Returns: int. A return value less than `ERO` indicates failure that can be interpreted as an `errno`, ≥ 0 indicates success, and a negative value indicates failure that can be interpreted as an `errno`.

TMEM_PUT_PAGE(pool_id, object_id, page_id, pfn)

Copy a page of data from a physical pageframe into a tmem pool and associate it with a handle-tuple.

Pool-id must identify a previously created pool, but object-id and page-id may represent either a new or existing page.

1 Any put (with one notable exception, see below) may be rejected and the client must be
2 prepared to deal with the failure.

3
4 After a page is put, the data exists both in the client pageframe and in the tmem pool. The
5 client is responsible for destroying or overwriting the client-side data, or alternately
6 managing any coherency between the copies.

7
8 Every page successfully put to a persistent pool must be found by a subsequent get using the
9 same handle-tuple (unless removed by a flush). A page successfully put to an ephemeral
10 pool has an indeterminate lifetime; even an immediately subsequent get may fail.

11 An implementation must enforce *put-put-get coherency*. For example, after the sequence:

```
12     tmem_put_page(ABC, D1);  
13     tmem_put_page(ABC, D2);  
14     tmem_get_page(ABC, E);
```

15
16 E may never contain the data from D1. For ephemeral pools, the implementation may
17 optionally flush the page on the second (duplicate) put, thus resulting in a failed get. For
18 persistent pools, a duplicate put must never fail, even if memory is unavailable.

19
20 Returns: int. Returns the number of pages successfully put (one or zero); a negative value
21 indicates failure that can be interpreted as an errno.

22
23
24 **TMEM_GET_PAGE(pool_id, object_id, page_id, pfn)**

25
26 *Lookup a page of data in tmem associated with a handle-tuple and, if found, copy it to a*
27 *physical pageframe.*

28
29 The handle-tuple must identify a previously put (or new'ed) page.

30
31 A successful get on a private+ephemeral pool is destructive, i.e. the copy is removed from
32 tmem as if a flush had also been performed. A successful get on any other pool type is non-
33 destructive.

34
35 Every page successfully put to a persistent pool must be found by a subsequent get using the
36 same handle-tuple (unless removed by a flush). A page successfully put to an ephemeral
37 pool has an indeterminate lifetime; even an immediately subsequent get may fail.

38
39 Regardless of pool type, the implementation must enforce *get-get coherency*. For example,
40 in the sequence

```
41     tmem_get_page(ABC, E);  
42     tmem_get_page(ABC, E);
```

43 if the first get fails, the second must also fail.

44
45 Returns: int. Returns the number of pages successfully got (copied into the pageframe),
46 which may be one or zero; zero is considered failure but not an error; a negative value
47 indicates failure that can be interpreted as an errno.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45

TMEM_FLUSH_PAGE(pool_id, object_id, page_id)

Disassociate a handle-tuple from any data in tmem

Pool-id must identify a previously created pool.

A flush on a valid pool_id must always succeed; that is, a get following a flush must fail.

Returns: int. Zero indicates there was no page to flush, 1 indicates the page was found and flushed. A negative return value indicates an invalid pool_id.

TMEM_FLUSH_OBJECT(pool_id, object_id)

Disassociate all pages associated with an object from any data in tmem

Pool-id must identify a previously created pool, but object-id and page-id may represent either a new or existing page (but see note 2). Pfn is a physical page frame number.

A flush object on a valid pool_id must always succeed; that is, a get to a page in this object following a flush object must fail.

Returns: int. TBD [is a return value necessary for any client?]. A negative return value indicates an invalid pool_id.

**TMEM_READ(pool_id, object_id, page_id, pfn,
 tmem_offset, pfn_offset, len)**

Lookup a page of data in tmem associated with a handle-tuple and, if found, copy a fixed number of bytes to a physical pageframe.

Pool-id must identify a previously created pool, and object-id and page-id must represent an existing page with the provided handle-tuple. The data sequence must be entirely contained within a page, i.e. offset+len must not exceed the pool's pagesize.

Returns: int. A return value ≥ 0 indicates success, and a negative value indicates failure that can be interpreted as an errno.

1 **TMEM_WRITE**(pool_id,object_id,page_id,pfn,
2 **tmem_offset,pfn_offset,len)**

3

4 *Copy a fixed number of bytes of data from a physical pageframe into an already existing*
5 *tmem pool page associated with a handle-tuple.*

6

7 Pool-id must identify a previously created pool, and object-id and page-id must represent an
8 existing page with the provided handle-tuple. The data sequence must be entirely contained
9 within a page, i.e. offset+len must not exceed the pool's pagesize.

10

11

12 Returns: int. A return value ≥ 0 indicates success, and a negative value indicates failure
13 that can be interpreted as an errno.

14

15

16 **TMEM_XCHG**(pool_id,object_id,page_id,pfn,
17 **tmem_offset,pfn_offset,len)**

18

19 *Exchanges a fixed number of bytes of data from a physical pageframe into an already*
20 *existing tmem pool page associated with a handle-tuple.*

21

22 Pool-id must identify a previously created pool, and object-id and page-id must represent an
23 existing page with the provided handle-tuple. The data sequence must be entirely contained
24 within a page, i.e. offset+len must not exceed the pool's pagesize.

25

26 [NOTE: needs more work, including some guarantee of synchronization]

27

28 Returns: int. A return value ≥ 0 indicates success, and a negative value indicates failure
29 that can be interpreted as an errno.

30

31

32 **TMEM_NEW_PAGE**(pool_id, object_id, page_id, pfn)

33

34 *Create an "empty page" in a tmem pool, initialize it with zeroes, and associate it with a*
35 *handle-tuple.*

36

37 [Not sure if this is really needed.]

38

39 Pool-id must identify a previously created pool, but object-id and page-id may represent
40 either a new or existing page.

41

42 Returns: int. A return value ≥ 0 indicates success, and a negative value indicates failure
43 that can be interpreted as an errno.

44

45

1 **TMEM_CONTROL (*tbd*)**
2
3 ***TBD***
4
5 [Not sure if this is really needed.]
6
7

8
9

10 **Appendix A.**

11

12 The client-to-implementation calling convention on x86 passes a pointer to a data structure.
13 The first item in this data structure is a 32-bit value that represents the operation to be
14 performed as follows:

15

TMEM CONTROL	0
TMEM NEW POOL	1
TMEM DESTROY POOL	2
TMEM NEW PAGE	3
TMEM PUT PAGE	4
TMEM GET PAGE	5
TMEM FLUSH PAGE	6
TMEM FLUSH OBJECT	7
TMEM READ	8
TMEM WRITE	9
TMEM XCHG	10

16