Intel Corporation: Intel Confidential

Pmemfile V1.0
Software Architecture Specification (SAS)

V.4

NVML based Linux SW Solution to Provide NVM Programming Model (NPM) Access to Persistent Memory via POSIX File System Interfaces.

1 CONTENTS

2 DOCUMENT REVISION HISTORY

Version	Document Changes
V.3 3/23/16	Initial document with completed high level interfaces, use cases, and theory of operations.
V.3 5/17/16	Made more changes to on media structures.
V.3 5/26/16	Additional modifications to on media format.
V.3 5/31/16	Additional changes and additions to on media format section. Modified Pmemfile_ Interception section to add more detail to the architecture chosen.
V.4 6/6/16	Completed initial persistent data structure definitions. Added open items section.
V.4 6/7/16	Finished persistent data structure design. Added file structure to volatile section.
V.4 6/8/16	More updates
V.4 6/14/16	Changes to the volatile and persistent structures. Added more details to the Pmemfile_ Intercept Library. Added diagram to show dependency resolution for lseek + open ().
V.4 10/3/16	Finalized volatile and persistent data structure layout. Modified interception library with new architecture.
V.4 10/17/16	Modified interception layer with new architecture.
V.4 10/26/16	Changes based on internal review with pmemfile team.
V.4 11/8/16	More changes based on internal review with pmemfile team
V.4 11/17/16	Finished persistent data structure section.
V.4 11/28/16	Updated based on review by pmemfile team
V.4 1/4/17	Updated persistent data structures
V.4 2/21/17	Added more detail to Section 7, design
V.4 3/14/17	Added more detail to valid states for persistent data structure members
V.4 3/16/17	Completed man pages and added links to document
V.4 3/21/17	Added Section 11.4, Consistency Checking
V.4 4/5/17	Modified High Level Architecture Diagram to be more correct and specific
V.4 4/10/17	Cleaned up for github publication.
V.4 4/19/17	More cleanup for github publication.

3 DOCUMENT OVERVIEW

This document describes the SW Architecture and High Level Design of the Pmemfile PMEM project.

4 TERMINOLOGY

This section outlines the major terminology and glossary items utilized throughout this architect specification. See the figure below for a graphical re-presentation of the terminology introduced in this terminology table.

Terminology, Glossary Term	Description
Persistent Memory, PMEM, CR-DIMM, NVDIMM	Byte addressable persistent memory.
NVM Programming Model(NPM)	SNIA model for software supporting non-volatile memory. http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.pdf
POSIX	A set of formal descriptions that provide a standard for the design of file systems.
System Call	Interface between user space and kernel space
(g)libc	Standard POSIX library interface for Unix operating systems

5 WHAT IS PMEMFILE?

Pmemfile is a PMEM (https://github.com/pmem) project which provides a POSIX –like file system interface to consumers while providing NVM programming model access to persistent memory. It is comprised of three parts: a) The preload library b) System Call Interception Library and c) The POSIX core library.

6 LINUX HIGH LEVEL ARCHITECTURAL COMPONENTS

6.1 HIGH LEVEL LINUX FILE INTERFACE

Applications in Linux use a high level interface to interact with files on a device. This interface is provided by the glibc library. The term "libc" is commonly used as a shorthand for the "standard C library", a library of standard functions that can be used by all C programs (and sometimes by programs in other languages). The standard functions provided by glibc include functions for file and file system access.

glibc is a user space library. However, since file management happens in the kernel it must communicate with the kernel to complete file operations. The interface glibc uses for communication with the kernel is the system call interface. A system call is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.

The following is a diagram which shows the high level layers from an application to the Linux kernel via glibc and the system call Interface.

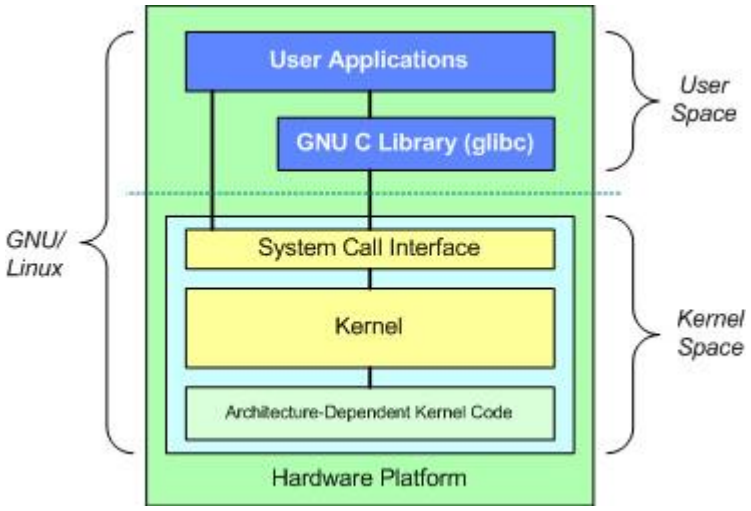


Figure 6-1 Linux Operating Environment

At the top is the user, or application, space. This is where the user applications are executed. Below the user space is the Linux kernel space.

7 PMEMFILE SW ARCHITECTURE

7.1 PMEMFILE ARCHITECTURAL COMPONENTS

Pmemfile will provide three architectural components: the Pmemfile preload library, system call interception library as well as the Pmemfile core library. The diagram shown in Figure 7-1 Pmemfile Architectural Components shows the components that will be delivered as part of Pmemfile as well as the high level interaction of these with other components on the system.

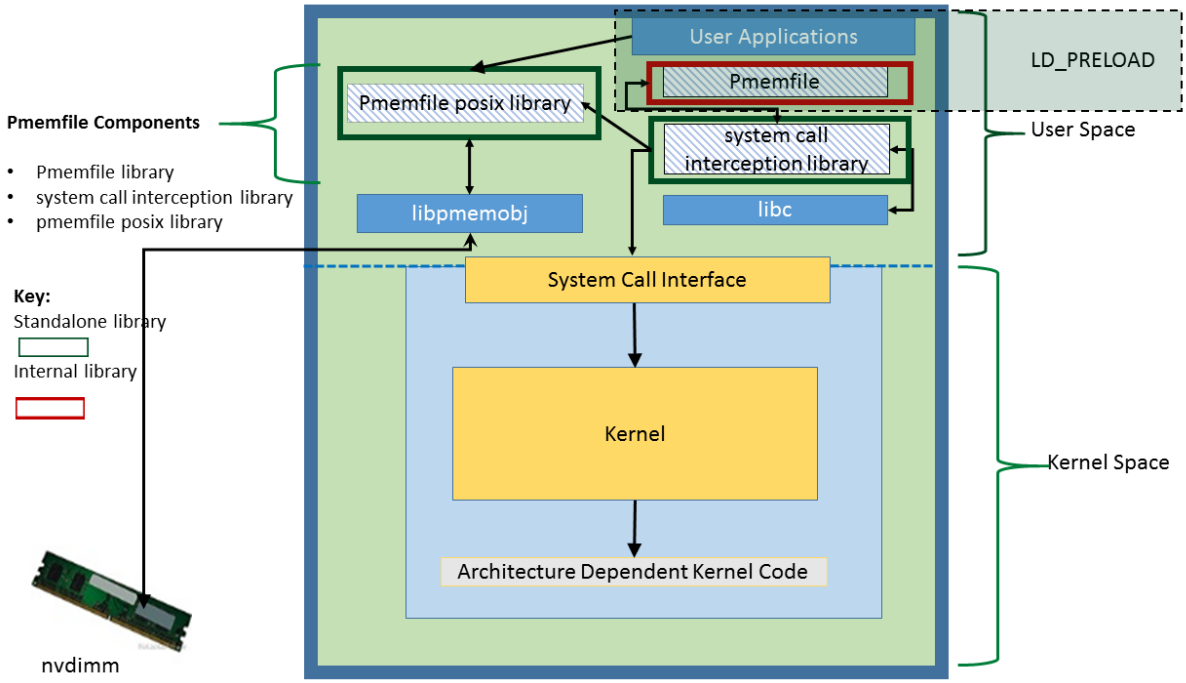


Figure 7-1 Pmemfile Architectural Components

7.1.1 Pmemfile Preload Library (libpmemfile.so)

The libpmemfile library implements the system call hook functions for the Pmemfile project. It uses the interface exported by the system call intercept library described in **SYSTEM CALL INTERCEPTION (LIBSYSCALL_INTERCEPT.SO)**. This library must be preloaded using the LD_PRELOAD linker directive when an application is started. When this library is preloaded into the application process space it will do the following:

- Create or open the Pmemfile pool
- Setup of hook functions for system call interception
- Pmemfile file descriptor management

7.1.1.1 Hook functions

The Pmemfile hook functions are wrappers that contain the logic for routing the system calls as defined in **SUPPORTED SYSTEM CALL INTERFACES.** If the file is a pmem-resident file the wrapper calls the appropriate Pmemfile POSIX library function. If not, it passes it through to the standard system call.

7.1.1.2 Pmemfile File Descriptor Management

During initialization a group of file descriptors is created from the kernel using '/dev/null' as the device. The pool of file descriptors are then mapped with a Pmemfile file as pmem resident files are created.

7.1.2 System Call Interception (libsyscall_intercept.so)

The Pmemfile interception library provides three major areas of functionality: 1) disassembling of glibc 2) building a trampoline table and 3) hot patching the file operation system calls in glibc. The System Call interception library is a public interface and can be used as a standalone component.

7.1.2.1 System Call Interception layer requirements and assumptions:

- Use of the system call interception layer along with the Pmemfile preload library will not require a re-compilation or re-link for the application.
- The system call interception layer will intercept file operation system calls prior to crossing the kernel boundary.
    - The system call interception layer will **only** cross the kernel boundary when absolutely required. It is never crosses the boundary to operate on the Pmemfile files. It is only used in preparation for access to the files. Currently, the following use cases require the use of the Linux kernel:
        - Getting a file descriptor for the Pmemfile file. We do this by opening /dev/null.
        - Path name resolution
- With Pmemfile each system call that is intercepted is made by glibc.
    - Any applications which call system calls directly will not be managed with the System Call interception layer.
- No other application or library tries to hot patch glibc in the same process.
- glibc must be initialized in the process address space prior to the System Call interception library.

7.1.2.2 System Call Interception Library Exported Interface

The Pmemfile interception library exports one interface:

```
int (*intercept_hook_point)(long syscall_number,
                            long arg0, long arg1,
                            long arg2, long arg3,
                            long arg4, long arg5,
                            long *result);
```

The consumer of this library has to define the hook functions for each of the system calls it wants to be intercepted.

All Linux system calls can take up to six arguments. Each system call is defined by a number. The *intercept_hook_point* interface provides a way to define a function which is called in lieu of the specified system call via the *syscall_number*.

7.1.2.3 System Interception Library Disassembling

Linux executable files, relocatable object files, core files and shared objects are all defined using the Executable and Linking Format (ELF). See man(5) elf.

The ELF format enables us to find locations of functions, data structures and other data types within a shared object such as glibc.

libmemfile disassembles glibc to find all the locations of the system calls Pmemfile supports as specified in **SUPPORTED SYSTEM CALL INTERFACES**. The Linux System Call interface is the fundamental interface between an application and the Linux kernel. Pmemfile does not use the kernel to do file operations however to maintain application compatibility Pmemfile intercepts at the system call layer, which in Linux is the supported and stable API.

Pmemfile uses the opensource project, Capstone, to disassemble glibc. The sections we must look in to find all instances of system calls are the '.text', '.symtab' and the '.dynsym' sections of the glibc library. The sections are disassembled, starting with .dynsym and .symtab. These sections hold the dynamic linking symbol table and the symbol table for the library, respectively. Some of the symbols found in these sections are functions in .text section, which means they are jump entry points. The .text section is then disassembled in full and the addresses of all system call instructions are stored, together with information about the instructions prior to and following the system call. A lookup table of all addresses which will be jump destinations is generated to help determine later whether an instruction is suitable for being overwritten -- of course, if an instruction is a jump destination it cannot be merged with the preceding instruction to create a new larger one.

Every time a system call instruction is found the next and previous padding bytes between routines are checked to find potential space for an extra jump instruction.

7.1.2.4 Pmemfile glibc Patching

Patching glibc is the process of modifying an address in glibc to point to a new address. The instruction at this new address is the one that is followed after the patch is applied. Pmemfile patching happens only after glibc is loaded in a process and does not affect the system wide glibc.

7.1.2.4.1 Memory Address Patching Considerations

To patch a system call located in glibc the following must be considered:

1) How do we ensure we never patch to an unknown or unexpected location within the process address space.
2) What is the distance between our patch function addresses and glibc in the process address space?
3) How many bytes are available to use for hot patching the code to insert the patch function.
4) How many bytes are over-written during the process of patching?
5) How do we patch the minimal number of bytes in glibc to achieve our goals
6) How do we go back to the original system call if we are not processing a file which is pmem resident.
7) How do we get back to glibc to complete the system call return?

7.1.2.4.1.1 Linux Address Space Layout

The first thing we must consider is the Linux Address Space Layout. It's important to understand this and to understand any restrictions or definitions that would affect how we ensure we never patch to an unknown or unexpected address within the process.

X86-64 provides a 64-bit address space for a process. Theoretically that is. In Linux there is a virtual hole in the middle of the address space so there is effectively 48 bits of available address. This is due to the fact that Intel processors support only 48-bit virtual address space. This means that any two libraries, or functions between them could be greater than 2GB apart. As a result we cannot simply use the relative address jump which only supports 32 bits which provides up to 2GB distance.

The linux x86-64 ABI explicitly states:

It cannot be assumed that a function is within 2GB in general. Therefore, it is necessary to explicitly calculate the desired address reaching the whole 64-bit address space.

Linux ABI Document, page 45:

http://refspecs.linuxfoundation.org/elf/x86-64-abi-0.99.pdf

7.1.2.4.1.2 Trampolining

Trampolining is a way to have an interim jump function that forwards on to the final jump location. In Pmemfile a trampoline table is created and mmaped to address that is within 2GB of glibc. The trampoline table contains wrapper functions that manage the routing of the system call to the appropriate component. Use of the Pmemfile trampoline table allows us to use the minimum number of bytes to patch glibc. This makes it easier to find adequate bytes to use for patching as well as making it less error prone. Once the wrapper function takes over the jumps can be a larger size since we are not constrained by the number of bytes we can overwrite.

The wrapper functions contain:

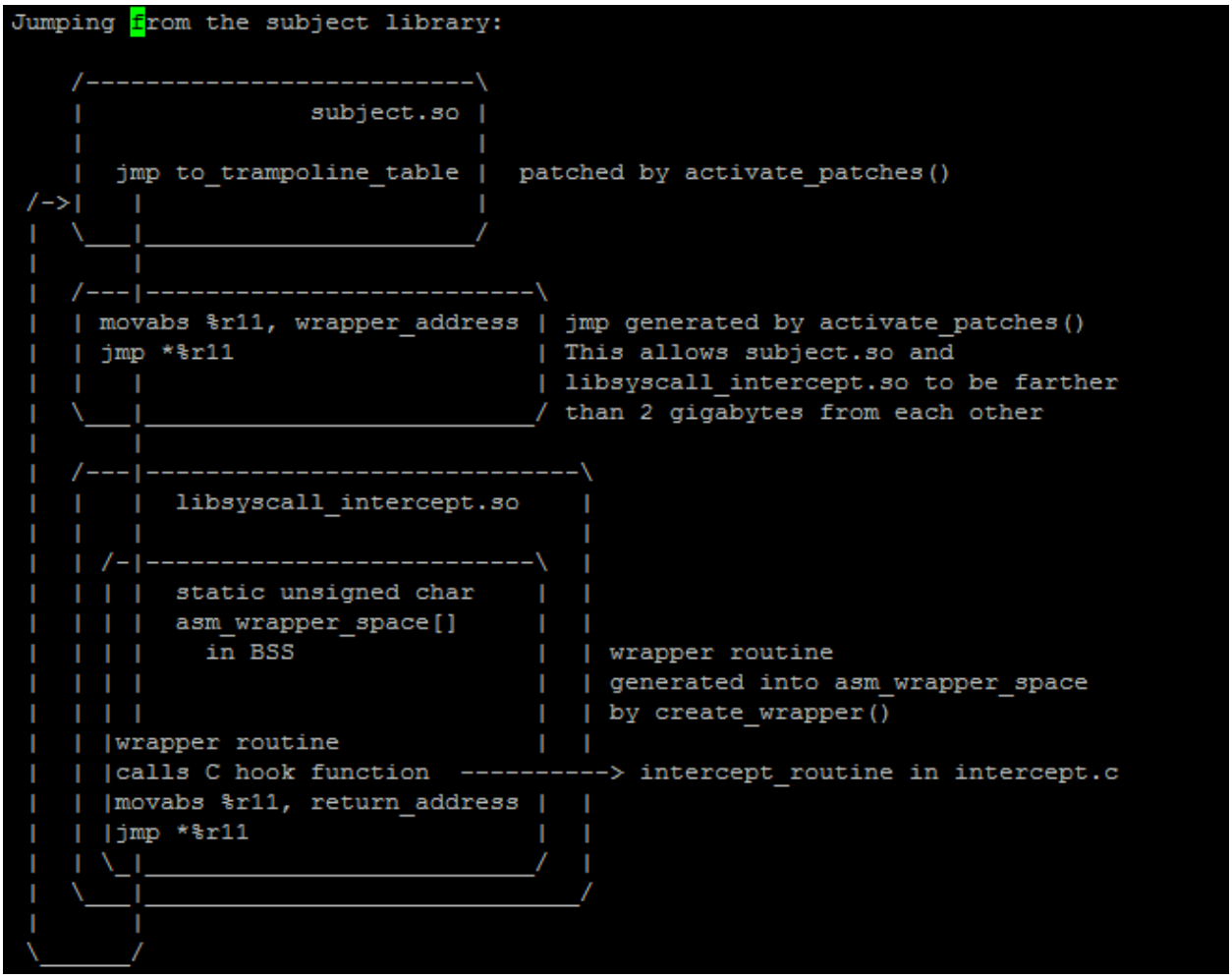
- Code to route file operation to correct component as per the hook functions defined
- Code which has return address into glibc for completion of the file operations
- Encapsulates the hook functions that were created in the preload phase

7.1.2.5 Pmemfile Interception Layer Mechanics

The steps to set up the Pmemfile interception layer are as follows:

- find the system calls in glibc
  - disassemble glibc and locate
- allocate trampoline table
- create patch wrappers
- activate patches
- Wrapper functions are created and the trampoline table is set up and glibc is disassembled.
-
- A call is made from the application to a glibc file operation
- The system call within the glibc function has a patch to jump to a location in the trampoline table.
- A jump is made from trampoline table to wrapper function
- Call to the C hook function (if pmem-resident file).
- Return to glibc to complete the system call.

The diagram below shows the high level operation of the Pmemfile system call library.



ERROR! REFERENCE SOURCE NOT FOUND.

7.1.3 Pmemfile POSIX Library (libpmemfile-posix.so)

The Pmemfile project provides a core library which encapsulates the file system operations in user space. For Linux the core library contains a POSIX interface library, libpmemfile-posix.so. For future development the core library component can also contain other interface support libraries.

The Pmemfile POSIX library provides a POSIX like interface that manages the Pmemfile file system. This library is a standalone component which can be used in conjunction with other NVML libraries and with or without the Pmemfile system call interception library. The Pmemfile POSIX library is the component of Pmemfile which provides Direct Access to the NVM media. It also provides consistency and correctness via the transactional interfaces provided the NVM libpmemobj library.

7.1.3.1 Pmemfile POSIX Library Requirements

- Must be usable in conjunction with other NVML libraries.

- Must provide an interface that is easy for application developers to use if they choose to modify applications rather than use the interception functionality. We have chosen, for Linux, to use POSIX as the guiding principle in designing the interfaces.
- The Pmemfile file system must look and behave as a 'normal' Linux file system providing the interfaces and behaviors defined in the **LIBPMEMFILE-POSIX.3 MANPAGE**. The Pmemfile POSIX library is the component that provides the 'look' and 'management' of the Pmemfile file system.
- Must provide the file system functionality in user space.
- Must provide at least basic Linux standard file system consistency guarantees.
- Must provide better performance, for certain workloads, than a DAX enabled file system.

7.1.3.2 Pmemfile POSIX Library Interfaces

An application developer can incorporate the Pmemfile POSIX library directly into an application. Rather than having to learn a new syntax the Pmemfile POSIX library provides, as much as possible, similar naming and similar function arguments as their POSIX function counterparts.

7.1.3.2.1 Pmemfile POSIX Library Interface Naming

The Pmemfile core library interfaces are named using the standard POSIX naming with 'Pmemfile' as the prefix. For example:

`open() /pmemfile_open()`

`close()/pmemfile_close()`

7.1.3.2.2 Exceptions to POSIX

In general a POSIX file operation expects a path or a file descriptor as the first argument. And when a file is being created or opened a file descriptor is returned. The Pmemfile POSIX library has some noted exceptions to this pattern.

7.1.3.2.2.1 Pmemfile POSIX Library Interface Signatures

In all cases the first argument of the libpmemfile-posix functions is a `` `PMEMfilepool' `` pointer. This is required since it is possible that an application could be working with multiple pools. We cannot rely on any specific behavior with regard to the number of pools.

In all cases where the POSIX call would require a standard file descriptor the Pmemfile POSIX Library functions will take a `'struct PMEMfile *'`. See Section File Descriptor Management for details.

7.1.3.2.2.2 Pmemfile POSIX Library Return Types

The return value for Pmemfile POSIX library functions which open and create files will always be a `'struct PMEMfile *'`. This structure is an opaque structure which represents, internally to libpmemfile-posix, the current information about a Pmemfile file. As a result of this any functions that modify or read files will take the returned `'struct PMEMfile *'` as input rather than a standard file descriptor.

7.1.3.2.2.3 File Descriptor Management

With Pmemfile when a call is made to the POSIX open() function, and the Pmemfile system call interface interception layer is present, a file descriptor for the Pmemfile file is generated by opening /dev/null. The Pmemfile system call interception layer then keeps track of these file descriptors and maps them to

the appropriate volatile Pmemfile file structure. If it is a non-pmem resident file it passes the open() call through to the system call interface which creates a file descriptor. It's important to note that the file descriptor given from opening /dev/null within the Pmemfile system call interface is a valid file descriptor from the system point of view. The file descriptors index into a per-process file descriptor table maintained by the kernel. This means that the Pmemfile file descriptors are also kept in the per-process file descriptor table in the kernel. The figure below shows the process of assigning file descriptors when the Pmemfile system call interface layer is present.

Choosing where to manage file descriptors is a critical aspect of Pmemfile. We chose to differentiate the 'file descriptors' between the Pmemfile system call interception layer and the Pmemfile POSIX library. As a result, the Pmemfile POSIX library interfaces do not take a standard file descriptor as an argument. This file descriptor for the Pmemfile POSIX library is a `'struct PMEMfile *'` which is described in Section **10.2.3**. A call to pmemfile_open() returns a `'struct PMEMfile *'` rather than a file descriptor. An application developer uses this in turn to perform operations on the file.

Why did we chose to have the management of file descriptors be handled in the Pmemfile system call interception layer?

- The way the Pmemfile POSIX library stores file data should be decoupled from the way the intercept layer stores it since the Pmemfile POSIX library is not dependent on the Pmemfile interception layer. The interception layer is a convenience provided.
- The Pmemfile system call interception layer always evaluates the ownership of a file and creates the file descriptor based on that ownership. It manages all file descriptors which are for pmem-resident files.
- If an application is using pmem-resident files managed by Pmemfile as well as standard media files within the same context the chance of errors goes up if file descriptors are created and managed by the Pmemfile POSIX library (this would mean a file descriptor would be a parameter into the Pmemfile POSIX functions as opposed to the `'struct PMEMfile *'`.
    - Even if a check was added to find the file descriptor in both the Pmemfile POSIX library and the kernel file descriptor table both would return true.
    - Remember: a Pmemfile file descriptor is also a valid non-pmem resident file descriptor due to the use of /dev/null.

8 SUPPORTED SYSTEM CALL INTERFACES

The system call intercept library defined in **SYSTEM CALL INTERCEPTION (LIBSYSCALL_INTERCEPT.SO)** supports the following system calls for Pmemfile.

The table indicates name, support provided, exceptions to existing system call behavior and notes on what the exception is. The **LIBPMEMFILE.1 MANPAGE** defines exceptions to support for each system call noted in the table below.

Table 1 Supported System Call Interfaces

Syscall	Supported	Exception	Exceptions noted
SYS_access	yes		
SYS_chdir	yes		
SYS_chmod	yes		
SYS_chown	yes		
SYS_chroot	no	yes	Pmemfile does not allow the process to change the root of the calling process. The root of the process is always the root of the Pmemfile file system.
SYS_clone	yes	yes	Clone is generally used to implement threads. An application must provide CLONE_THREAD as a flag otherwise the command will fail. The following flags are not supported: CLONE_IO CLONE_NEWIPC CLONE_NEWNET CLONE_NEWNS CLONE_NEWPID CLONE_NEWUTS CLONE_PARENT and related flags CLONE_PID CLONE_VFORK This flag is supported: CLONE_VM: However, even if a thread is created with shared virtual memory the child will not be able access, create or modify any pmem-resident files.
SYS_close	yes		
SYS_creat	yes	yes	See open
SYS_dup	yes	yes	When dup() is called the refcount on the Pmemfile handle in libpmemfile will be increased to keep track of number of references to a file.
SYS_dup2	no	yes	Same as dup()
SYS_dup3	yes	yes	Close on exec is always set Same rule as for dup, RE: refcount.
SYS_epoll_ctl	no	yes	We are not supporting polling of any kind at this time.
SYS_epoll_pwait	no	yes	Same as above
SYS_epoll_wait	no	yes	Same as above

SYS_execve	yes	yes	Pmemfile does not support execve when the executable file is a Pmem-resident file. If the 'filename' value is a pmem-resident file this will return an ENOEXEC.
SYS_faccessat	yes		
SYS_fadvise64	no	yes	Pmemfile does not support fadvise64.
SYS_fallocate	yes		
SYS_fchdir	yes		
SYS_fchmod	yes		
SYS_fchmodat	yes		
SYS_fchown	yes		
SYS_fchownat	yes		
SYS_fcntl	yes	yes	Pmemfile support fcntl with the following flag exceptions: Duplicating File Descriptors F_DUPFD_CLOEXEC Pmemfile always sets this flag. Since this flag is always set the libpmemfile layer will return success when consumer sets this. It will not go through interception layer. File Descriptor Flags F_SETFD The only flag supported is O_CLOEXEC. Pmemfile always sets this flag. Handling the same as DUPFD_CLOEXEC. File Status F_SETFL Is supported. O_ASYNC - never, O_DIRECT - always, O_NONBLOCK – ignored In call cases a 0 will be returned. F_GETFL is supported.

			Locking F_SETLK, F_SETLKW, F_GETLK, F_UNLCK Are supported. F_SETOWN, F_GETOWN_EX, F_SETOWN_EX Not supported. Will return EINVAL. F_GETSIG, F_SETSIG Not supported. Will return EINVAL. F_SETLEASE, F_GETLEASE Not supported. Will return EINVAL F_NOTIFY Not supported. Will return EINVAL MANDATORY LOCKS Not supported. Will return EINVAL. LEASES Not supported. Will return EINVAL. File and Directory change notifications Not supported. Will return EINVAL. Pipe manipulation Not supported. Will return EINVAL. File sealing Not supported. Will return EINVAL.
SYS_fdatasync	yes	yes	All IO with Pmemfile is synchronous. Libpmemfile component will return 0 for this call.
SYS_fgetxattr	no	yes	No xattrs support provided.
SYS_flistxattr	no	yes	Same as above
SYS_flock	no	yes	Pmemfile does not support file locking. Will return EINVAL.
SYS_fork	yes	yes	Pmemfile does not provide multi-process support. A child created with fork() will not be able to access any existing pmem-resident files nor create new ones.
SYS_fremovexattr	no	yes	No xattrs support
SYS_fsetxattr	no	yes	Same as above
SYS_fstat	yes		
SYS_fstatfs	yes		
SYS_fsync	yes	yes	Libpmemfile component will return 0 for all file sync commands.
SYS_ftruncate	yes		
SYS_ftruncate64	yes		
SYS_futimesat	yes		
SYS_getcwd	yes		
SYS_getdents	yes	yes	This does not have a glibc wrapper. The real support is for readdir(). Man page will reflect the support of readdir().
SYS_getdents64	yes	yes	Same as above

SYS_inotify_add_watch	no	yes	Pmemfile does not support any I/O event notification
SYS_inotify_rm_watch	no	yes	Same as above
SYS_io_cancel	No	yes	Pmemfile will not support asynchronous IO for V1.0
SYS_io_destroy	no	yes	Same as above
SYS_io_getevents	no	yes	Same as above
SYS_io_setup	no	yes	Same as above
SYS_io_submit	no	yes	Same as above
SYS_ioctl	yes*	yes	*Pmemfile V1.0 will not provide support for this.
SYS_lchown	yes		
SYS_lgetxattr	no	yes	No xattrs support provided
SYS_link	yes		
SYS_linkat	yes		
SYS_listxattr	no	yes	Same as all xattrs
SYS_llistxattr	no	yes	Same as all xattrs
SYS_lremovexattr	no	yes	Same as all xattrs
SYS_lseek	yes		
SYS_lsetxattr	no	yes	Same as all xattrs
SYS_lstat	yes		
SYS_mkdir	yes		
SYS_mkdirat	yes		
SYS_mknod	no	yes	Pmemfile does not support block or character devices
SYS_mknodat	no	yes	Same as above
SYS_mmap	no?	yes	If Yes – mmap supports mmap for the following flags: MAP_SHARED, MAP_FIXED Protection flags supported: PROT_READ and PROT_WRITE is the default
SYS_newfstatat	yes		
SYS_open	yes	yes	Pmemfile supports open with the following flag exceptions: O_ASYNC - EINVAL O_CNOTTY – Always enabled Only supported for terminals, pseudo terminals, sockets and fifo's Pmemfile does not support these devices. O_DIRECT – Always enabled. O_DSYNC – Always enabled. O_NONBLOCK – Ignored. O_SYNC – Always enabled. O_CLOEXEC – Always enabled.

SYS_open_by_handle_at	yes*	yes	*Not in V1.0
SYS_openat	yes		
SYS_poll	no	yes	Pmemfile does not support any event driven file management
SYS_ppoll	no	yes	Same as above
SYS_pread/64	yes		
SYS_pselect	no	yes	Pmemfile does not support any event driven file management.
SYS_pwrite64	yes		
SYS_pwritev	yes		
SYS_read	yes		
SYS_readahead	no	Yes	Is not supported. Pmemfile does not support caching as it always operates in direct access mode.
SYS_readlink	yes		
SYS_readlinkat	yes		
SYS_readv	yes		
SYS_removexattr	no	yes	Pmemfile does not support xattrs
SYS_rename	yes	yes	Pmemfile does not support renaming files between Pmemfile file systems.
SYS_renameat	yes	yes	Same as above
SYS_renameat2	no	yes	
SYS_rmdir	yes		
SYS_select	no	yes	Same as pselect
SYS_setxattr	no	yes	Same as other xattrs support
SYS_sendfile	yes*	yes	Pmemfile supports this if both in fd and out fd are pmem-resident files. *Not supported in V1.0
SYS_splice	yes*	yes	Pmemfile supports this if both the out fd and in fd are pmem-resident files. * Not supported in V1.0
SYS_stat	yes		
SYS_statfs	yes		
SYS_swapoff	no	yes	Pmemfile cannot be used as a swap device
SYS_symlink	yes		
SYS_symlinkat	yes		
SYS_sync	yes	yes	Pmemfile treats this as a no-op since all IO is synchronous.
SYS_sync_file_range	yes	yes	Pmemfile treats this as a no-op. Pmemfile IO is always synchronous.
SYS_syncfs	yes	yes	Same as above
SYS_sysfs	yes		
SYS_truncate	yes		

SYS_unlink	yes		
SYS_unlinkat	yes		
SYS_utime	yes		
SYS_utimensat	yes		
SYS_utimes	yes		
SYS_vfork	no	yes	Pmemfile does not support this call.
SYS_write	yes		
SYS_writev	yes		

9 LIBPMEMFILE-POSIX INTERFACE AND SYSTEM CALL SUPPORT

The libpmemfile-posix library will not provide support for all of the system calls defined in Section **SUPPORTED SYSTEM CALL INTERFACES.**

Specifically, libpmemfile-posix does not provide support for:

- Thread or process creation. This is handled by the calling application.
  - A child process cannot create or access any file in the current Pmemfile pool. The top level libpmemfile interface will manage handling and verification of this.
- Any data or file system sync calls. All operations in Pmemfile are synchronous.
- No advisement operations such as readahead or fadvise. This does not make sense in the Pmemfile space.

The **LIBPMEMFILE-POSIX.3 MANPAGE** defines and describes all behaviors for the interfaces exported by the core library.

10 VOLATILE AND PERSISTENT DATA STRUCTURES

10.1 PMEMFILE DATA STRUCTURE CONSIDERATIONS

There are some high level architectural decisions that have been made for reducing wear on the persistent media.

1. Locks that protect persistent data structure fields are defined in the volatile data structure. This is because lock values change frequently and can cause unnecessary wear on the media.
2. The naming of the volatile data structures that have an on-media counterpart will have the type prefixed by a 'v', e.g. `pmemfile_vinode`.

10.2 PMEMFILE VOLATILE DATA STRUCTURES

The Pmemfile volatile data structures will not be specifically defined here as they are subject to change and are not a contract between the user and the Pmemfile library. However, the general principles and behavior of these is described in this section.

10.2.1 PMEMfilepool

The pmemfile_pool structure is the top level libpmemfile-posix data structure. It provides a pointer to the pool as well as a pointer to the pool super block.

10.2.2 pmemfile_vinode

The pmemfile_vinode structure provides access to the on-media inode, locking for the on-media inode as well as keep track of the reference count. Locking for an inode is not required to be persistent. If the system crashes or when the file system closes all file descriptors are destroyed and no locks are held on the on media inode.

The pmemfile_vnode structure also store any caches related to an inode. For example, block mapping, directory file lookup.

10.2.3 pmemfile_file

The pmemfile_file structure a handle to an opened file. This pmemfile_file structure will be mapped with a file descriptor that is generated when the file is opened. In general this data structure contains:

- current offset information
- locks
- pointer to the volatile inode for this file
- file flags

10.3 PERSISTENT DATA STRUCTURES

Persistent data structures are those stored on the persistent media and describe the file system metadata. We must strive to ensure that the format of the persistent data structures are complete and stable in their definition. However, it's possible we may have to make changes moving forward which may require a version change. Locking is provided in the volatile data structures defined for each persistent structure unless otherwise specified.

10.3.1 pmemfile_super

The superblock is a unique data structure in a filesystem. It is the top level data structure which describes high level metadata about the pmemfile file system.

```
struct pmemfile_super {
      /* Superblock version */
      uint64_t version;

      /* Root directory inode */
      TOID(struct pmemfile_inode) root_ino;

      /* List of arrays of inodes that were deleted, but are still
      opened */
      TOID(struct pmemfile_inode_array) orphaned_inodes;

      char pad[4096
                - 8 /* version */
                - 16 /* toid */
                - 16 /* toid */];
};
```

10.3.1.1 Valid states for pmemfile_super members

- version > 0
- root_ino != NULL
- orphaned inodes != NULL
    - All deleted files and directory inodes are put on the orphaned list when a file is deleted. Inodes on this list are cleaned up when pmemfile_vinode->ref == 0 and pmemfile_inode->nlink == 0;
- Pad is initialized to 0 and not used at this point.

10.3.2 pmemfile_block

```
struct pmemfile_block {
      TOID(char) data;
      uint32_t size;
      uint32_t flags;
      uint64_t offset; /* offset from the beginning of the file */
      TOID(struct pmemfile_block) next; /* next block with data */
      TOID(struct pmemfile_block) prev; /* prev block with data */
};
```

10.3.2.1 Valid states for pmemfile_block members

- data = pointer to valid data address
- size = Multiple of 4k, variable size
- flags = initialized
- offset >= 0. Offset into file
  next, prev = NULL or valid pointer to next block with data block && do not belong to another inode

10.3.3 pmemfile_block_array

The block array is per inode.

```
struct pmemfile_block_array {
      TOID(struct pmemfile_block_array) next;

      /* size of the blocks array */
      uint32_t length;
      uint32_t padding;
      struct pmemfile_block blocks[];
};
```

10.3.3.1 Valid states for pmemfile_block_array members

- next = NULL or TOID of next member in array
- length = number of possible blocks in array
- blocks[] = array of pmemfile_block structures
- Holey files:
  - Holes in a file are not represented as blocks.
  - A hole is detected when blocks[n]->next.offset != blocks[n].offset + blocks[n].size

10.3.4 pmemfile_dirent

Directory entries are individual members of a parent directory.

```
struct pmemfile_dirent {
           TOID(struct pmemfile_inode) inode;
           char name[PMEMFILE_MAX_FILE_NAME + 1];
};
```

10.3.4.1 Valid states for pmemfile_dirent

- inode != NULL && must be an inode in use(nlink > 0)
- name != NULL && null terminated and no longer than PMEMFILE_MAX_FILE_NAME + 1

10.3.5 pmemfile_dir

Directory entries are stored in a directory object. Multiple directory entries are contained in a directory object.

```
struct pmemfile_dir {
      uint32_t num_elements;
      uint32_t padding;
      TOID(struct pmemfile_dir) next;
      struct pmemfile_dirent dentries[];
};
```

10.3.5.1 Valid states for pmemfile_dir

- num_elements >= 2. At least '.' and '..'
- next == NULL or TOID of next directory
- dentries[] != NULL. Must have '.' and '..', in that order

10.3.6 pmemfile_time

```
struct pmemfile_time {

  /* Seconds */

  int64_t sec;

  /* Nanoseconds */

  int64_t nsec;

};
```

10.3.6.1 Valid pmemfile_time members

- sec an nsec should be non-negative.
- nsec < 10 the nth

10.3.7 pmemfile_inode

A pmemfile_inode represents the file metadata. All file types, directories and regular files, have a pmemfile_inode representation.

```
struct pmemfile_inode {
        /* Layout version */
        uint32_t version;

        /* Owner */
        uint32_t uid;

        /* Group */
        uint32_t gid;

        uint32_t reserved;

        /* Time of last access. */
        struct pmemfile_time atime;

        /* Time of last status change. */
        struct pmemfile_time ctime;

        /* Time of last modification. */
        struct pmemfile_time mtime;

        /* Hard link counter. */
        uint64_t nlink;

        /* Size of file. */
        uint64_t size;

        /* File flags. */
        uint64_t flags;

        /* Data! */
        union {
                /* File specific data. */
                struct pmemfile_block_array blocks;

                /* Directory specific data. */
                struct pmemfile_dir dir;

                char data[4096
                                - 4  /* version */
                                - 4  /* uid */
                                - 4  /* gid */
                                - 4  /* reserved */
                                - 16 /* atime */
                                - 16 /* ctime */
```

```
                                        - 16 /* mtime */
                                        - 8  /* nlink */
                                        - 8  /* size */
                                        - 8  /* flags */];
        } file_data;
};
```

10.3.7.1 Valid states for pmemfile_inode

- Version > 0
- uid, gid – valid uid and gid as defined on system
- atime, ctime, mtime – valid pmemfile_time data
- nlink > = 1, orphaned inodes nlink can be 0
    - Directory inodes have nlinks = 2 + number of subdirectories
- size = regular file = length of file, dir = number of bytes a directory takes, symlinks length of the path
- flags = flags + mode
    - See: Supported System Call Interfaces, open() for flag and mode exceptions
- file_data:
    - blocks
        - Mutually exclusive from dir
        - List will be non-NULL if regular file && blocks cannot be specified in more than one inode
    - dir
        - Mutually exclusive from blocks. Blocks for directory are kept in the inode for directory. If dir != NULL then this inode is a directory.
    - char data[4096-size of current members of pmemfile_inode]
        - This is used as storage for the symlink path if a symlink

10.3.8 pmemfile_inode_array

```
struct pmemfile_inode_array {
	PMEMmutex mtx;
	TOID(struct pmemfile_inode_array) prev;
	TOID(struct pmemfile_inode_array) next;

	/* Number of used entries, <0, NUMINODES_PER_ENTRY> */
	uint32_t used;

	char padding[12];
	TOID(struct pmemfile_inode) inodes[NUMINODES_PER_ENTRY];
};
```

10.3.8.1 Valid states for pmemfile_inode_array

- mtx != NULL when modifying inode arrays or used value, used when we unlink file. Pmemobj will clean up.
- prev, next, prev != NULL, next == NULL or valid array object.
- used = number of non-null entries in array.
- pad is zeroed at initialization. Unused.
- inodes – entries in this inode array can be null. Only inserted at unlink. Number of non-null entries in this area == used

10.4 MODEL FOR CHECKING CONSISTENCY OF METADATA ON MEDIA

At any point in time the on-media data structures must be in a consistent state.

10.4.1 Consistency Checking

For pmemfile V1.0 we will not have a consistency checker. The use of the transactions offered in libpmemobj provide a level of consistency that is sufficient for the first release. Section **PERSISTENT DATA STRUCTURES** defines all valid values for each member of the persistent data structure. This section describes in more detail how we plan to provide consistency checking when an 'fsck' like tool is developed. This section is intended to aid developers and system administrators in understanding the valid states of a pmemfile file system.

10.4.1.1 Pmemobj object consistency

The integrity of the pmemobj pool is the first consistency check that will be done. Tools such as pmemcheck and pmempool can be utilized to verify the object consistency within the pool.

10.4.1.2 Pmemfile object consistency

All on media Pmemfile data structures are pmemobj objects. This section describes, at a high level what a consistency checker will do for Pmemfile.

10.4.1.2.1 Inodes

The first step in checking for Pmemfile consistency will be to process all inodes:

The following information will be gathered and cached in support of checking inode consistency:

- list of directory inodes
- list of file inodes
- list of inodes in use(nlink >0)
- list of blocks in use
- list of duplicate blocks
- list of inodes with bad fields(as noted above)
- list of inodes which are symlinks
- list of orphaned inodes

The inode fields are checked:

- mode field is legal
- size and block count fields are correct
- Blocks are not in use by another inode
- Broken symlinks (no file on the other end of the symlink)

10.4.1.2.2 Directories inodes

Verification of directories includes:

- That a directory entry has at least two entries, '.' and '..'
- '.' is the first entry and it's inode should be that of the directory
- '..' is the 2nd entry in the directory and it's inode is that of the parent directory
- The directory inode number should refer to an in use inode.

  - Not on orphaned list
- The directory entry name is a null terminated string and is not larger than PMEMFILE_MAX_FILE_NAME + 1 in size.
- The number of elements in the dirents[] structure is the same as the value for num_elements.

10.4.1.2.3 Directories and directory entries

- Finds file system root directory and marks it as processed.
- Then, for each directory it attempts to trace up the file system tree until it reaches a directory that has been previously processed. This previously processed parent directory means that it has been found to be attached to the file system root.
- If we cannot get to a previously verified root directory of the path we are currently traversing we mark this directory as detached and the following data is stored:
  - inode number
  - permissions
  - owner, group
  - size
  - date

10.4.1.2.4 Super Block

Validate:

- There is a super block object for the Pmemfile file system.
- The values for the super block are valid:
  - Valid root inode
  - Valid version

10.4.1.2.5 Repair

Repair of a file system can be managed, in some cases, without user intervention. The following issues can be repaired without intervention:

- No root inode
  - To do this all directories must be attached except to the root inode which is missing. This means all directories below the root inode must be marked as processed.
- Broken symlinks
  - Symlink inode will be deleted
- Duplicate blocks
  - File will be cloned, using current block data and multiple block reference inodes will be deleted
- Orphaned inode list
  - Process list, if inode is not on media but in orphaned inode list, just remove from list
  - Otherwise, process as normal
- File size
- Some inode flags and modes
- Missing ‘.’ or ‘..’ entries if all other directory data is clean

10.4.1.2.6 When repair is not possible

In the following cases automated repair will not be offered:

- Detached directories.
    - If root inode but not all directories are attached to a path which leads to root
- Corrupted inode fields
- Corrupted/incorrect versions

11 Error Handling

11.1 Error Handling

Pmemfile will return valid errors for all operations supported. The system call interface layer will return valid errno's for each corresponding call. The Pmemfile POSIX library will set errno as appropriate and return errors as defined in the man page linked in Section **libpmemfile-posix.3 manpage.**

12 Appendix A

12.1 libpmemfile.1 manpage

libpmemfile.1.txt

12.2 libpmemfile-posix.3 manpage

libpmemfile-posix.3.txt