Good Old-Fashioned Persistent Memory

TERENCE KELLY

Terence Kelly studied computer science at Princeton and the University of Michigan, earning his PhD at the latter in 2002. He then spent 14 years at Hewlett-Packard Laboratories. During his final five years at HPL, he developed software support for non-volatile memory. Kelly now teaches and evangelizes the persistent memory style of programming. His publications are listed at http://ai.eecs.umich.edu/~tpkelly/. tpkelly@eecs.umich.edu

Byte-addressable non-volatile memory (NVM)—Intel Optane—is now shipping in volume. Today's NVM offers performance between that of DRAM memory and flash storage [2, 7] and can be accessed via either storage or memory interfaces [8]. The latter offers the prospect of radically simplifying application software by allowing direct manipulation of persistent data via CPU instructions (LOAD and STORE), thus offering an alternative to traditional persistence technologies such as relational databases and key-value stores. Industrial adoption of NVM and its corresponding style of programming is growing [9].

Given the excitement surrounding novel NVM hardware, now is a good time to remind ourselves that it has long been possible to implement a software abstraction of persistent memory ("p-mem") on *conventional* hardware—ordinary volatile DRAM and block-addressed durable storage devices. The corresponding "p-mem style of programming" resembles the style that NVM invites, and supports similar simplifications, but doesn't require special NVM hardware.

This article illustrates p-mem programming on conventional hardware with C code for UNIX-like operating systems; all code is available at [3]. Spoiler alert: the basic technique is to lay out application data in memory-mapped files, with help from a few easy tricks and patterns. Because conventional `mmap()` doesn't guarantee data integrity in the face of failures, crash consistency requires extra support. The right crash consistency mechanism for p-mem programming on conventional hardware is *failure-atomic `msync()`* (FAMS) [6], and this article presents a concise new implementation of FAMS.

A Persistent Linked List

The C program below prepends words from `stdin` to a persistent singly linked list. It relies on a bare-bones persistent memory library, `pmem`, presented later. Notice that the list node data structure's `next` field is not a conventional pointer but rather an *offset*—specifically a `pmo_t` ("persistent memory offset type"), defined as a `uintptr_t` in `pmem.h`. Under the hood, `pmem` computes offsets relative to the base address where persistent data are mapped, which may vary on different runs of the program. Offsets allow data structures to be *relocatable*, which improves portability and facilitates sharing persistent data between different applications. The alternative of *non*-relocatable persistent data offers different tradeoffs and is beyond the scope of this article; see [5] for a discussion.

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include "pmem.h"

typedef struct {
  pmo_t next;
  char string[];
} node_t;

#define NP(o) ((node_t *)pmem_o2p(o))
```

```
int main(int argc, char *argv[]) {
  int r;
  char buf[100];  /* harmonize with scanf() below */
  pmo_t head, t;
  if (2 != argc) {
    fprintf(stderr, "usage: %s pmemfile\n", argv[0]);
    return 1;
  }
  if (0 != (r = pmem_map(argv[1]))) {
    fprintf(stderr, "pmem_map() failed: %d\n", r);
    return 2;
  }
  head = pmem_get_root();
  while  (1 == scanf(" %99s", buf)) {
    if (0 == strcmp("[dump]", buf))
      for (t = head; 0 != t; t = NP(t)->next)
      printf("%s\n", NP(t)->string);
    else {
      t = pmem_alloc(sizeof(node_t) + 1 + strlen(buf));
      if (0 == t) {
       fprintf(stderr, "pmem_alloc() failed\n");
       return 3;
      }
      strcpy(NP(t)->string, buf);
      NP(t)->next = head;
      head = t;
      pmem_set_root(head);
    }
  }
  return 0;
}
```

Function `pmem_map()` maps a given persistent data file into memory, initializing persistent heap metadata within the file if necessary. Unlike conventional `mmap()`, `pmem_map()` returns an error code rather than the address where the file has been mapped. Clients of `pmem` (i.e., code that uses `pmem`) neither know nor care about persistent data addresses—that's the whole point of relocatability. Clients allocate from a persistent heap in the file via `pmem_alloc()`, which returns offsets rather than conventional `malloc()`'s pointers. Finally, the `pmem` library embeds a *root offset* within the persistent data file. Clients must ensure that all persistent data are reachable from the root by calling `pmem_set_root()`. This allows the client to obtain an entry point into persistent data structures via `pmem_get_root()` on subsequent executions. Our list example program maintains the invariant that the root offset is always the head of the persistent linked list.

Function `pmem_o2p()` converts offsets to conventional pointers, which macro `NP` casts to a list-node pointer. Clients of `pmem` need offset-to-pointer conversions for accessing the innards of application-defined data structures. However, the `pmem` library doesn't support *pointer-to-offset* conversions because well-designed applications don't need them: clients' persistent data structures should contain only offsets returned by `pmem_alloc()` (or offsets derived therefrom), never pointers; only offsets are encountered when traversing persistent data.

The shell commands below demonstrate that our program's list is indeed persistent. `truncate` creates a new sparse backing file whose size is a multiple of the system page size. We run `list` twice, feeding it different words and dumping the list. The second dump shows that the words entered on the first run have persisted.

```
% truncate -s 409600 list.bf
% echo 'wun too [dump]' | ./list list.bf
too
wun
% echo 'free fore [dump]' | ./list list.bf
fore
free
too
wun
```

Persistent memory programming based on memory-mapped files is much more versatile and powerful than the brief examples above would suggest. In particular, retrofitting persistence onto legacy software that was not designed for persistence can be remarkably easy, and the rules governing multithreaded p-mem are straightforward [5].

Library Internals

The `pmem` library interfaces used above admit a succinct no-frills implementation, shown below. There's nothing arcane going on; much of the code simply checks internal consistency and catches corner-case errors, syscall failures, and client misuse. The library often returns line numbers where errors occur rather than `errno`-like codes ("use the Source, Luke"), and the persistent heap supports a p-mem allocator but no corresponding `free()`.

The `pmem` library defines a header structure (`pmh_s`) that will occupy the first few machine words of the backing file that contains persistent data. The header contains allocator bookkeeping information and the root offset described above. The library stores in static external variables `e_base` and `e_len`, respectively, the address at which the backing file is mapped and the size of the backing file.

Library function `pmem_map()` invokes conventional `mmap()` to map a specified backing file into the caller's address space; at most one such mapping at a time is supported. Function `pmem_unmap()` removes mappings; `pmem_alloc()` allocates persistent memory; and the paired `pmem_[get|set]_root()` functions provide access to the root offset.

Again, the pmem library is intentionally Spartan. It serves merely to remind us that a few dozen lines of code suffice to support rudimentary persistent memory programming on conventional hardware.

```
#include <assert.h>
#include <fcntl.h>
#include <stdint.h>
#include <stddef.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "pmem.h"

static_assert(sizeof(pmo_t) == sizeof(void *),   /* C11 */
              "offsets & pointers incompatible");

typedef struct {  /* header of backing file & in-memory image */
  pmo_t avail, end,  /* allocator bookkeeping */
        root;        /* live data must be reachable from root */
} pmh_s;  /* "persistent memory header structure" */

static pmh_s * e_base;  /* start address of in-memory image */
static size_t  e_len;     /* length of in-memory image */

#define UNIT (_Alignof(max_align_t))  /* C11 */
#define ALIGNED(o) (0 == (o) % UNIT)
#define ALIGN(o) do { while(! ALIGNED(o)) (o)++; } while (0)

/* Backing file and its in-memory image consist of a header (of
   type pmh_s above), a heap (nearly everything else), and final
   padding (one UNIT). Padding at the high end eliminates an
   awkward corner case. A root offset must "point" within the
   heap. Other offsets (e.g., "end") may point one byte beyond
   heap, analogous to C rule for pointers (N1570 Sec 6.5.6). */
#define VALID(o)  \
  (0 == (o) || (sizeof *e_base <= (o) && (o) <= e_len - UNIT))
#define VALID_ROOT(o) \
  (0 == (o) || (sizeof *e_base <= (o) && (o) <  e_len - UNIT))

#define SANITY_CHECKS    \
  do {                   \
    assert((NULL == e_base && 0 == e_len) ||                    \
           (NULL != e_base && 0 != e_len) );                    \
    assert(NULL == e_base ||                                    \
           (  ALIGNED(e_base->avail) && ALIGNED(e_base->end)    \
            && VALID(e_base->avail) &&   VALID(e_base->end)     \
            && VALID_ROOT(e_base->root) ));                     \
  } while (0)

void * pmem_o2p(pmo_t o) {  /* convert offset to pointer */
  assert(VALID(o));
  return 0 == o ? NULL : (char *)e_base + o;
}

#define P20(p) ((pmo_t)((char *)(p) - (char *)e_base))

#define RL return __LINE__  /* indicates where error occurs */

int pmem_map(const char * const file) {
  int fd, prot = PROT_READ | PROT_WRITE, flag = MAP_SHARED;
  long int pgsz;  struct stat sb;  size_t s;  pmh_s *t;
  SANITY_CHECKS;
  if (NULL != e_base)  /* limit: one mapping at a time */      RL;
  if (1 > (pgsz = sysconf(_SC_PAGESIZE)))                      RL;
  if (UNIT > (size_t)pgsz)                                     RL;
  if (0 > (fd = open(file, O_RDWR)))                           RL;
  if (0 != fstat(fd, &sb))                                     RL;
  if (10 * UNIT + sizeof *t > (s = (size_t)sb.st_size))        RL;
  if (0 != s % (unsigned long)pgsz)                            RL;
  if (MAP_FAILED ==
      (t = (pmh_s *)mmap(NULL, s, prot, flag, fd, 0)))      {
    if (0 != close(fd))  /* don't leak fds ... */              RL;
    else                                                   RL; }
  if (0 != close(fd))                                     {
    if (0 != munmap(t, s))  /* ... or memory either */        RL;
    else                                                   RL; }
  /* file must be either new or already initialized: */
  if ( ! (   (0 == t->avail && 0 == t->end && 0 == t->root)
          || (0 != t->avail && 0 != t->end)))                  RL;
  if (! (ALIGNED(t->avail) && ALIGNED(t->end)))                RL;
  e_base = t;
  e_len  = s;
  if (! (VALID(t->avail) && VALID(t->end)
         && VALID_ROOT(t->root)))                              RL;
  if (0 == t->avail) {  /* initialize persistent heap */
    t->avail = P20(1 + t);
    ALIGN(t->avail);
    t->end   = P20((char *)t + s - UNIT);
    t->root  = 0;
  }
  else  /* previously initialized; check size: */
    if (t->end != P20((char *)t + s - UNIT))                   RL;
  SANITY_CHECKS;
  return 0;
}

pmo_t pmem_alloc(size_t n) {  /* "bump-pointer" allocator */
  pmo_t r;
  SANITY_CHECKS;
  assert(NULL != e_base);
  if (0 == n                                || /* ask 0, get 0 */
      e_base->avail    >= e_base->end       || /* out of p-mem */
      e_base->avail    >  ~(pmo_t)0 - n     || /* "+n" overflows */
      e_base->avail + n >  e_base->end)       /* <n bytes left */
    return 0;
  r = e_base->avail;
  e_base->avail += n;
```

```
  ALIGN(e_base->avail);
  SANITY_CHECKS;
  return r;
}

int pmem_unmap(void) {
  SANITY_CHECKS;
  if (NULL == e_base)             RL;
  if (0 != munmap(e_base, e_len))  RL;
  e_base = NULL;
  e_len = 0;
  return 0;
}

void pmem_set_root(pmo_t o) {
  SANITY_CHECKS;
  assert(NULL != e_base && VALID_ROOT(o));
  e_base->root = o;
}

pmo_t pmem_get_root(void) {
  SANITY_CHECKS;
  assert(NULL != e_base);
  return e_base->root;
}
```

Crashes and Data Integrity

Could a full-featured incarnation of the `pmem` library be suitable for serious purposes? Yes, for applications that always perform an orderly shutdown. However, `pmem` is inadequate for applications that must tolerate sudden crashes, e.g., power outages, OS kernel panics, and application software crashes. Why? Because `pmem` creates shared file-backed memory mappings with conventional `mmap()`, which cannot prevent crashes from corrupting the backing file. One fundamental problem is that the OS may write modified memory pages down to the backing file at any time and in any order, regardless of if/when `msync()` is called. Another problem is that if `msync()` *is* called, the changes it makes to the backing file are not atomic with respect to failure. The state of the backing file following a crash is therefore indeterminate.

Failure-atomic `msync()` (FAMS) solves this problem by strengthening the semantics of conventional `mmap()` / `msync()`. FAMS guarantees that the backing file always reflects the most recent successful `msync()`, regardless of failures [6]. The FAMS abstraction is the ideal foundation for crash-tolerant persistent memory programming on conventional hardware. It has been implemented in the Linux kernel, in file systems, and in user-space libraries; at least six FAMS implementations exist, two of which are in commercial products [5]. FAMS has the attractive property that underlying durable storage is a freely configurable placeholder: "durability" for a FAMS-based p-mem program can mean anything from a single hard disk to a RAID array or geo-replicated cloud storage. Furthermore, FAMS is easy to reason about because it merely *restricts* the behavior of well-understood standard interfaces: FAMS guarantees behavior that is possible (but, sadly, unlikely) in conventional `mmap()` / `msync()`.

Existing FAMS implementations have demonstrated the abstraction's power and versatility, but they're not without barriers to adoption: some are research prototypes, others are buried in appliance-like commercial products, and the two newest implementations are complex but not yet thoroughly tested [4, 5]. The world needs a FAMS implementation that is efficient enough for serious use yet simple enough to audit easily.

Simple and Efficient Crash Consistency

Our efficient yet very simple new userspace implementation of failure-atomic `msync()` makes two compromises: it restricts our choice of file system, and its interface is fussier than classic FAMS.

The library implementation below is called `famus_snap` ("failure-atomic `msync()` in userspace via snapshots"). It runs on file systems that allow multiple files to share physical storage, e.g., Btrfs, XFS, and OCFS2 (optionally accessed over a network via NFSv4.2 or CIFS). The interesting work happens in function `famus_snap_sync()`, which uses `ioctl(FICLONE)` to create a new snapshot that shares storage with the backing file. A copy-on-write mechanism ensures that subsequent modifications to one file do not affect the other.

The interfaces of both the `mmap()` and `msync()` analogs require the caller to supply a file descriptor for an empty write-only snapshot file. When these functions return successfully, the snapshot file contains the current state of the backing file and is read-only. Whereas post-crash recovery in classic FAMS uses the backing file, recovery in `famus_snap` *replaces* the backing file with the most recent readable snapshot file. As a side effect, `famus_snap` gives us data versioning for free: every snapshot is a version of the backing file, which may be retained indefinitely or deleted to reclaim storage resources.

```
#define _POSIX_C_SOURCE 200809L

#include <stddef.h>
#include <unistd.h>
#include <linux/fs.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include "famus_snap.h"

static int rwperm(mode_t m, unsigned int r, unsigned int w) {
  return (!!(m & S_IRUSR) == r) && (!!(m & S_IWUSR) == w);
}
```

```
#define  L __LINE__
#define RL return L  /* indicates where error occurs */

/* We must fsync() backing file twice to ensure that snapshot
   data are durable before success indicator (file permission)
   becomes durable. We're not using fallocate() to reserve
   space for worst-case scenario in which backing file and
   snapshot file diverge completely, because that could defeat
   the reflink sharing that makes snapshots efficient; read
   "man ioctl_ficlone". The "ioctl(FICLONE)" works only on
   reflink-enabled file systems, e.g., Btrfs, XFS, OCFS2. */
int famus_snap_sync(fd_t bfd, fd_t snapfd, fd_t dirfd) {
  struct stat sb;
  if (0 != fstat(snapfd, &sb))                  RL;
  if (! rwperm(sb.st_mode, 0, 1))               RL;
  if (0 != ioctl(snapfd, FICLONE, bfd))         RL;
  if (0 != fsync(snapfd))                       RL;
  if (0 != fchmod(snapfd, S_IRUSR))             RL;
  if (0 != fstat(snapfd, &sb))                  RL;  /* paranoia */
  if (! rwperm(sb.st_mode, 1, 0))               RL;  /* paranoia */
  if (0 != fsync(snapfd))                       RL;
  if (0 < dirfd && 0 != fsync(dirfd))           RL;
  if (0 != close(snapfd))                       RL;
  return 0;
}

#define RN return NULL

void * famus_snap_map(void * addr, size_t * plen, int flags,
                      fd_t bfd, fd_t snapfd, fd_t dirfd,
                      int * status) {
  struct stat sb;  void *a;  int prot = PROT_READ | PROT_WRITE;
  if (NULL == status)               {             RN; }
  if (NULL == plen)                 { *status = L; RN; }
  if (0 == (flags & MAP_SHARED))    { *status = L; RN; }
  if (0 != fstat(bfd, &sb))         { *status = L; RN; }
  *plen = (size_t)sb.st_size;
  a = mmap(addr, *plen, prot, flags, bfd, 0);
  if (MAP_FAILED == a)    { *status = L; RN; }
  if (NULL == a) {
    if (0 != munmap(a, *plen))        *status = L;
    else                              *status = L;
    RN;
  }
  if (0 != (*status = famus_snap_sync(bfd, snapfd, dirfd))) {
    if (0 != munmap(a, *plen))        *status = L;
    RN;
  }
  return a;
}
```

The full source code for `famus_snap` is available at [3]. It requires a reflink-capable file system such as Btrfs, XFS, or OCFS2. If you're eager to run `famus_snap` but you don't have such a file system handy, consider installing one *within a file* on some other file system; just run the following commands as `root`:

```
# truncate --size 512m XFSfile
# mkfs.xfs -m crc=1 -m reflink=1 XFSfile
# mkdir XFSmountpoint
# mount -o loop XFSfile XFSmountpoint
# xfs_info XFSmountpoint
# cd XFSmountpoint
  [run famus_snap test...]
```

Streamlined Implementation

The `famus_snap` library above is a reasonably efficient way to implement failure-atomic `msync()` in userspace. However, with an in-kernel implementation like the prototype posted by Christoph Hellwig [1], similar semantics can be implemented more efficiently by taking advantage of the mechanisms that the XFS file system uses to implement the reflink system call.

In that case the existing code path to allocate new blocks and write them out of place when overwriting data is used independently of the B-tree tracking reference counts for blocks shared after using the reflink system call. In this case in addition to the actual block allocation, only the special records that ensure that the blocks are cleaned up when recovering from an unclean shutdown are required. This ensures the overhead of the write is similar to that for extending a file or filling a hole, but the extra overhead for manipulating block reference counts is avoided.

Conclusion

Persistent memory programming on conventional hardware is possible, thanks to `mmap()` and a few tricks that don't get as much attention as they deserve. Regardless of whether conventional hardware or newfangled NVM is available, the great advantage of the p-mem style of programming is simplicity—readers skeptical on this point are invited to re-write the persistent linked list program above using, e.g., a relational database or key-value store for persistence.

For crash-tolerant applications, failure-atomic `msync()` provides precisely the right fortified semantics for `mmap()`-based p-mem programming. The new FAMS implementation presented in this article is concise, clear, and thus easy for readers to audit because it leverages efficient file snapshotting from userspace. Christoph Hellwig's implementation in XFS achieves greater efficiency by avoiding unnecessary work. Until NVM supplants DRAM, FAMS can support crash-safe p-mem programming on conventional hardware.

Acknowledgments
Christoph Hellwig reviewed the snapshot-based implementation of failure-atomic `msync()`, suggested the procedure for creating a quick XFS installation within a different file system, provided a description of his FAMS implementation for XFS, and supplied information about reflink-capable file systems.

References

[1] C. Hellwig, "Failure Atomic Writes for File Systems and Block Devices": https://lwn.net/Articles/715918/.

[2] J. Izraelevitz, J. Yang, L. Zhang, A. Memaripour, Y. J. Soh, S. R. Dulloor, J. Zhao, J. Kim, X. Liu, Z. Wang, Y. Xu, S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," April 2019: https://arxiv.org/abs/1903.05714v1.pdf.

[3] T. Kelly, Example code to accompany this article: https://www.usenix.org/sites/default/files/kelly_code.tgz.

[4] T. Kelly, "`famus`: Failure-Atomic `msync()` in User Space": http://web.eecs.umich.edu/~tpkelly/famus/.

[5] T. Kelly, "Persistent Memory Programming on Conventional Hardware," *ACM Queue*, vol. 17, no. 4, July/August 2019: https://queue.acm.org/detail.cfm?id=3358957.

[6] S. Park, T. Kelly, K. Shen, "Failure-Atomic `msync()`," in *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 225–238: https://dl.acm.org/citation.cfm?id=2465374.

[7] I. B. Peng, M. B. Gokhale, E. W. Green, "System Evaluation of the Intel Optane Byte-addressable NVM," International Symposium on Memory Systems (MemSys), Sept. 2019: https://memsys.io/.

[8] A. Rudoff, "Persistent Memory Programming," *;login:*, vol. 42, no. 2, Summer 2017: https://www.usenix.org/system/files/login/articles/login_summer17_07_rudoff.pdf.

[9] S. Swanson (organizer), Persistent Programming in Real Life (PIRL) [conference], 2019: https://pirl.nvsl.io/.