pynvm Documentation

Release 0.4.dev

Christian S. Perone

Dec 29, 2017

Contents

This framework aims to bring NVM (*non-volatile memory*)/SCM (*storage-class memory*) technology functionality to the Python ecosystem. The PyNVM package provides an 'nvm' namespace that contains several sub-packages. These packages wrap the sub-modules provided by PMDK: Persistent Memory Development Kit. Most of the sub-modules are relatively thin wrappers around the corresponding PMDK API calls, with some scaffolding to make them more easily used from Python. The `pmemobj` submodule provides a fully Pythonic interface to persistent memory, allowing the easy persistence of Python objects via a `PersistentObjectPool`.

Contents:

CHAPTER 1

Introduction

This library provides Python bindings for PMDK: Persistent Memory Development Kit.

1.1 Overview and Rationale

Currently, there are no Python packages supporting *persistent memory*, where by *persistent memory* we mean memory that is accessed like volatile memory, using processor **load** and **store** instructions, but retaining its contents across power loss just like traditional storages.

The goal of this project is to provide Python bindings for the libraries that are part of PMDK: Persistent Memory Development Toolkit. The **pynvm** project aims to create bindings for PMDK without modifying the Python interpreter itself, thus making it compatible to a wide range of Python interpreters (including PyPy).

1.2 How it works

In the image above, we can see different types of access to a NVDIMM device. There are the standard and well known types of access like the one using the standard file API (fopen/open, etc.). Then on the far right is the type of access that this package is concerned with, using Load/Store and bypassing all kernel space code. This is the fastest way an application can access memory, and in our case, this is not the traditional volatile memory, it is **persistent memory**. The significance of this is that you don't need to serialize data to disk anymore to save it between program runs, you just keep your data structures in memory that is **persistent**.

Providing the infrastructure to do this reliably is the purpose of Intel's PMDK, for which this package provides Python bindings.

See also:

Planning the Next Decade of NVM Programming.

Programming Models for Emerging Non-Volatile Memory Technologies.

Persistent Memory Byte-Addressable Non-Volatile Memory.

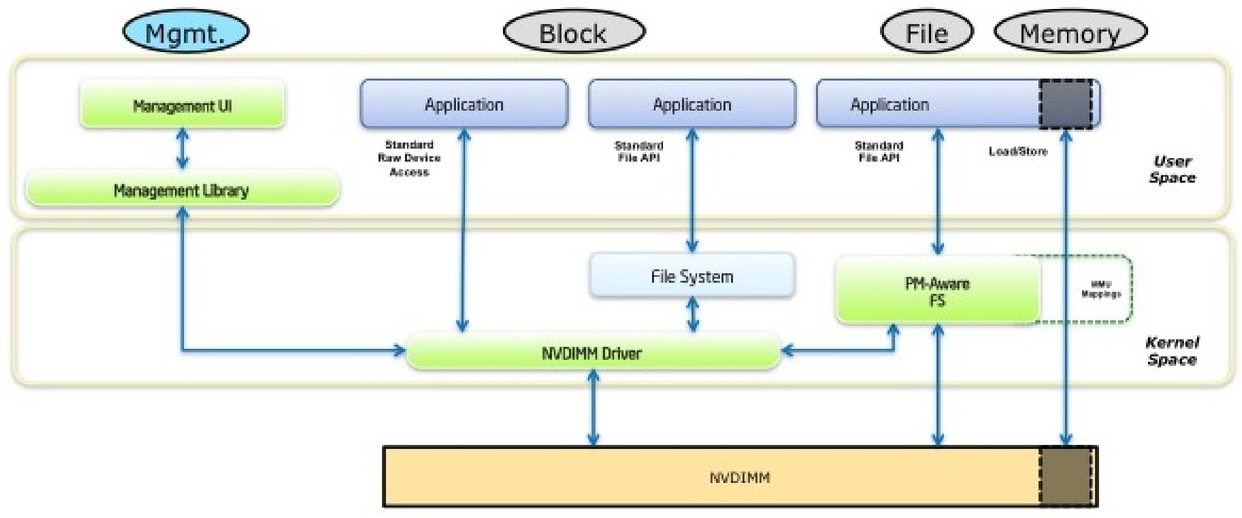

Fig. 1.1: *Image from: http://pmem.io*

Persistent Memory: What's Done, Coming Soon, Expected Long-term.

CHAPTER 2

Getting Started

2.1 Installation and Requirements

To install **pynvm** you need to first install PMDK:

- PMDK (install instructions at Github)

You can then install **pynvm** using **pip**:

```
pip install pynvm
```

pip will automatically install all dependencies for the Python package. To use **pynvm** in your program, import the submodule you wish to use via the `nvm` namespace. For example, to use pmemobj you can do:

```python
import nvm.pmemobj
```

2.2 Using pmem (*low level persistent memory*)

The *`nvm.pmem`* module provides an interface to the PMDK libpmem API, which provides low level persistent memory support. This module provides tools you can use to implement tear-proof persistent memory updates, but working at this level your application is solely responsible for protecting against tears.

See also:

For more information regarding **libpmem**, please refer to the libpmem documentation.

Here is an example of opening a PMEM file, writing to it, and reading from it:

```python
import os
from nvm import pmem
from fallocate import posix_fallocate

# (optional) check the pmem library version
pmem.check_version(1, 0)
```

```
# Open file to write and fallocate space
fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# mmap it using pmem
reg = pmem.map(fhandle, 4096)

# Write on it and seek to position zero
reg.write("lol" * 10)
reg.write("aaaa")
reg.seek(0)

# Read what was written
print(reg.read(10))
print(reg.read(10))

# Persist the data into the persistent memory
# (flush and hardware drain)
pmem.persist(reg)
```

Here is an example of using context managers for flush and drain and numpy buffers:

```
import os
import numpy as np
from nvm import pmem
from fallocate import posix_fallocate

fhandle = open("dst.dat", "w+")
posix_fallocate(fhandle, 0, 4096)

# Will persist (pmem_persist) and unmap
# automatically
with pmem.map(fhandle, 4096) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

    # This will create a numpy array located at
    # persistent memory (very cool indeed) where you
    # can reshape as you like
    n = np.frombuffer(reg.buffer, dtype=np.int32)
    print(n.shape)

# Flush context will only flush processor caches, useful
# in cases where you want to flush several discontiguous ranges
# and then run hardware drain only once
m = pmem.map(fhandle, 4096)
with pmem.FlushContext(m) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

# Will only execute the hardware drain (if available)
m = pmem.map(fhandle, 4096)
with pmem.DrainContext(m) as reg:
    reg.write("lol" * 10)
    reg.write("aaaa")

fhandle.close()
```

2.3 Using pmemlog (*pmem-resident log file*)

The `nvm.pmemlog` module provides an interface to the PMDK libpmemlog API, which provides pmem-resident log (*append-only*) file memory support. Writes to the log are atomic.

See also:

For more information regarding the **libpmemlog**, please refer to the libpmemlog documentation.

Here is an example of creating a persistent log, appending a record to it, and printing out the logged record:

```
from nvm import pmemlog

# Create the logging and print the size (default is 2MB when not
# specified)
log = pmemlog.create("mylogging.pmemlog")
print(log.nbyte())

# Append to the log
log.append("persistent logging!")

# Walk over the log (you can also specify chunk sizes)
def take_walk(data):
    print("Data: " + data)
    return 1

log.walk(take_walk)
# This will show: "Data: persistent logging!"

# Close the log pool
log.close()
```

2.4 Using pmemblk (*arrays of pmem-resident blocks*)

The `nvm.pmemblk` module provides an interface to the PMDK libpmemblk API, which provides support for arrays of pmem-resident blocks. Writes to the blocks are atomic.

See also:

For more information regarding the **libpmemblk**, please refer to the libpmemblk documentation.

Here is an example of creating a block pool and writing into the blocks:

```
from nvm import pmemblk

# This will create a block pool with block size of 256 and
# 1GB pool
blockpool = pmemblk.create("happy_blocks.pmemblk", 256, 1<<30)

# Print the number of blocks available
print(blockpool.nblock())

# Write into the 20th block
blockpool.write("persistent block!", 20)

# Read the block 20 back
data = blockpool.read(20)
```

```
blockpool.close()

# Reopen the blockpool and print 20th block
blockpool = pmemblk.open("happy_blocks.pmemblk")
print(blockpool.read(20))

blockpool.close()
```

2.5 Using pmemobj (*persistent objects*)

The `nvm.pmemobj` module provides an interface to the PMDK libpmemobj API, which provides transactionally managed access to memory that supports allocating and freeing memory areas. In this case, rather than providing a simple wrapper around the pmemobj API, which by itself isn't very useful from Python, pynvm provides a full Python interface. This interface allows to you store Python objects persistently.

This is a work in progress: currently persistence is supported only for lists (PersistentList), dicts (PersistentDict), objects (PersistentObject), integers, strings, floats, None, True, and False. This is, however, enough to do some interesting things, and an example (pminvaders2, a port to python of the C example) is included in the examples subdirectory.

Here is an example of creating a PersistentObjectPool and storing and retrieving objects:

```
from nvm import pmemobj

# An object to be our root.
class AppRoot(pmemobj.PersistentObject):
    def __init__(self):
        self.accounts = self._p_mm.new(pmemobj.PersistentDict)

    def deposit(self, account, amount):
        self.accounts[account].append(amount)

    def transfer(self, source, sink, amount):
        # Both parts of the transfer will succeed, or neither will.
        with self._p_mm.transaction():
            self.accounts[source].append(-amount)
            self.accounts[sink].append(amount)

    def balance(self, account):
        return sum(self.accounts[account])

    def balances(self):
        for account in self.accounts:
            yield account, self.balance(account)

# Open the object pool, creating it if it doesn't exist yet.
pop = pmemobj.PersistentObjectPool('myaccounts.pmemobj', flag='c')

# Create an instance of our AppRoot class as the object pool root.
if pop.root is None:
    pop.root = pop.new(AppRoot)

# Less typing.
accounts = pop.root.accounts

# Make sure two accounts are created.  In a real ap you'd create these
# accounts with subcommands from the command line.
for account in ('savings', 'checking'):
    if account not in accounts:
        # List of transactions.
        accounts[account] = pop.new(pmemobj.PersistentList)
        # Starting balance.
        accounts[account].append(0)

# Pretend we have some money.
pop.root.deposit('savings', 200)

# Transfer some to checking.
pop.root.transfer('savings', 'checking', 20)

# Close and reopen the pool.  The open call will fail if the file
# doesn't exist.
pop.close()
pop = pmemobj.PersistentObjectPool('myaccounts.pmemobj')

# Print the current balances.  In a real ap this would be another
# subcommand, run at any later time, perhaps after a system reboot...
for account_name, balance in pop.root.balances():
    print("{:10s} balance is {:4.2f}".format(account_name, balance))

# You can run this demo multiple times to see that the deposit and
# transfer are cumulative.
```

CHAPTER 3

`pmemobj` tutorial

`pmem`, `pmemblk`, and `pmemlog` will be interesting to people with specific needs that match the services provided by those libraries. The vast majority of Python programmers interested in utilizing persistent memory, however, will be interested in `pmemobj`, which provides a Python-object oriented interface to persistent memory. This chapter aims to explain how to use Python `pmemobj`, and how to write your own `Persistent` objects.

3.1 Conceptual Overview

In a normal Python program we create a bunch of objects and use them to accomplish a goal. When the program ends, all of the objects are thrown away, to be rebuilt from scratch the next time the program is run. `pmemobj` provides the opportunity to change this paradigm: to be able to create objects in a program, and have their state preserved between program runs, so that they do not need to be reconstructed the next time the program is run. The guarantee made by `pmemobj` is that the state of the objects will be self-consistent no matter when the program terminates. Further, it provides a `transaction()` that can be placed around multiple persistent object modifications to guarantee that either *all* of the modifications are made, or *none* of them are made.

Contrast this with a persistence paradigm such as that provided by SQL Alchemy. Here we have objects whose data is mapped to relational database tables. When the program starts up, it can query the database in any of several ways in order to retrieve objects. The object state is thus persistent in the sense that an object will have the same state it had the last time that object was flushed to disk in a previous program. SQLAlchemy also provides transactions that guarantee that either all of the changes in a block are committed to the database, or none of them are.

So, how do the two paradigms differ? At the higher conceptual levels, not by much. In the SQLAlchemy case objects are retrieved by running a query to find selected instances of a given object class. In `pmemobj` objects are retrieved by walking an object tree from a `root` object defined by the program. The difference, for both better and worse, is that persistent memory is entirely an "object store", and *not* a relational database. It is thus more similar to the ZODB than to SQLAlchemy.

Where it differs from the ZODB is in how objects are stored. In the ZODB Python objects are serialized using the `pickle` module and stored on disk. In `pmemobj`, objects are stored directly in persistent memory, written to and read from using the same *store* and *fetch* instructions used to access RAM memory. This means that in principle read access can be nearly as fast as RAM access, and write access can be orders of magnitude more efficient than disk writes.

In practice we're at the early stages of development, and at least in the Python case we aren't anywhere near as fast as we could be. But it's fast enough to be useful.

To be a bit more concrete, consider the example of a Python list. CPython stores a list in RAM via an object header that points to an area of allocated memory that holds a list of pointers to the objects in the list. In `pmemobj`, a list is stored in *persistent* memory as an object header that points to an area of allocated *persistent* memory that contains a list of *persistent* pointers to the objects in the list. An access to a list element is a normal `addr+offset` fetch of a pointer. Pointer resolution is another quick arithmetic operation. Updating a list element is the reverse: calculating the persistent pointer to the object and storing it at the correct offset in the persistent data structure. It is clear that this is going to be more efficient than SQLAlchemy marshalling to SQL-DB-update to disk-write to disk-flush, or ZODB-pickling to disk-write to disk-flush.

There is, however, overhead involved in the integrity guarantees. `libpmemobj` uses a change-log to record all changes that are taking place in a transaction, and if the transaction is aborted or not marked as complete, then all of the changes that did take place during the aborted transaction are rolled back, either immediately in the case of an abort, or the next time the persistent memory is accessed by `libmemobj` in the case of a crash. This log overhead has a non-zero cost, but what you buy with that cost is the object and transactional integrity in the face of hard crashes. And all of the log and rollback activity takes place using direct memory *fetch* and *store* instructions, so it is still fast, relatively speaking.

In this first version of `pmemobj` we have focused on proof of concept and portability rather than efficiency. That is, it is implemented entirely in Python, using CFFI to access the `libpmemobj` functions. In addition, most immutable persistent objects are handled by converting them back to normal Python RAM based instances when accessed, rather than accessing them directly in persistent memory. All of this adds conceptually unnecessary overhead and results in execution times that are slower than optimal. There is no conceptual barrier, however, to making it all quite efficient by moving the object access to the C level in a future version. The object algorithms are, for the most part, copied directly from the CPython codebase, with a few modifications to deal with persistent pointers and updating the rollback log. So in principle the object implementations can be almost as fast as the CPython objects they are emulating.

3.2 Real and Fake Persistent Memory

"Real" persistent memory in the context of this library is physical non-volatile memory that is accessible via the linux kernel DAX extensions. Persistent memory thus configured appears as a mounted filesystem to Linux. An allocated area of persistent memory is labeled by a filename according to normal unix rules. Thus if your DAX memory is mounted at /mnt/persistent, your would refer to an allocated area of memory named `myprog.pmem` via the path:

> /mnt/persistent/myprog.pmem

The persistent file system is a normal unix filesystem when viewed through the file system drivers. The magic of DAX, however, is that it allows a program to bypass the file system drivers and have direct, unbuffered access to the memory using normal CPU *fetch* and *store* instructions. There are, of course, concerns with respect to CPU caches and when exactly a change gets committed to the physical memory. See the `pmem` module for more details. `pmemobj` handles all of those details so your program doesn't have to.

There are two sorts of "fake" persistent memory. One is discussed on the Persistent Memory Wiki referenced above: you can emulate real persistent memory using regular RAM by reserving RAM to accessed through DAX via kernel configuration.

The second sort of "fake" persistent memory is to simply `mmap` a normal file. In this case the pmem libraries use different calls to ensure changes are flushed to disk, but the remainder of the pmem programming infrastructure can be tested. All of the pmem libraries automatically use this mode when the specified path is not a DAX-backed path.

So, anywhere in the following examples where a filename is used, you can substitute a path that will access the fake or real persistent memory as you choose, and the examples should all work the same. (Except for losing the persistent data on machine reboot, if you are using RAM emulation.)

3.3 Object Types and Persistence

For the purposes of considering persistence, we can divide Python objects up into three classes: immutable non-container objects, mutable non-container objects, and container objects.

Immutable non-container objects are the easiest to handle. We can store them in whatever form we want in persistent memory, and upon access we can reconstruct the equivalent Python object and let the program use that. Because the object is immutable, it doesn't matter that the object in persistent memory and object in use aren't the same object. (Or if it does, that's a bug in your program, since Python makes no guarantees about the identity of immutable objects.)

Mutable non-container objects *must* directly store, update, and retrieve their data from persistent memory, since everything that points to that mutable object will expect to see any updates. (An example of a mutable non-container object is a `bytearray`. *pmemobj* does not yet support any of Python's mutable non-container types.)

Container objects may contain pointers to other objects. The rule in *pmemobj* is that every object pointed to by a persistent container must itself be stored persistently. This means that all pointers inside persistent objects are persistent pointers; that is, pointers that can be resolved into a valid pointer if the program is shut down and restarted running in a different memory location. Therefore we can't map a persistent immutable container object (such as a tuple) to its Python equivalent, because the stored pointers are persistent pointers, and may not even have the same length as a normal RAM pointer.

Mostly these distinctions matter only to someone implementing a new `persistent` type. However, the first category, the immutable non-container objects, matter at the Python programming level. This is because there are two possibilities for such objects: *pmemobj* may support them directly, or it may support them through `pickle`. If a class is supported directly, a `Pesistent` container may reference them and *pmemobj* will automatically deal with storing their data persistently, and accessing it when referenced. If a class is not supported directly, then a program using *pmemobj* can still reference them, if the program nominates them for persistence via pickling. This is less efficient than direct support, but allows programs to use data types for which support has not yet been written. (Pickling is not applied automatically because there is no way for *pmemobj* to determine if a specific class is immutable or not.)

3.4 Hello <your_name_here>

We'll start the tutorial proper with the traditional "Hello, World" program. To make it interesting from a persistence standpoint, we'll skip past the static "Hello, world!" to the second part of the traditional example, where you make it say hello to a specified name, and we'll make it remember the name from one call to the next:

```
from nvm.pmemobj import PersistentObjectPool

with PersistentObjectPool('hello_world.pmem', flag='c') as pool:
    if pool.root is None:
        name = input("What is your name? ")
        pool.root = name
    print("Hello, {}".format(pool.root))
```

This simple example demonstrates several things. Persistent memory is accessed through a `PesistentObjectPool` object. By passing `flag='c'` to the constructor, we tell *pmemobj* to create the pool if it doesn't exist yet, and to open it if it does. It creates the pool with a fairly generous size, but a real application might need to increase the allocated size depending on how much data it is handling.

Note that a pool's size is fixed once created. There are plans for future improvements that will either provide a way to resize a pool or, at a minimum, a way to dump the data from one pool and restore it into another. Neither of these facilities exist as of this writing.

The pool object returned by the constructor has several methods and one attribute. That attribute, *root*, names an arbitrary persistent object, and its default value is `None`. When a pool is first created, then, `root` is `None`. Our program checks if `root` is `None`, and if it is, sets about getting a value (the name to use). It assigns that to the `root` attribute, which is enough to cause that object to be persisted. It then prints out the "Hello" greeting.

If `root` is not `None`, then it has a value, so we use that value to print out the greeting.

If we name this script `hello.py`, running it from the command line would look like this:

```
> python hello.py
What is your name? David
Hello, David
> python hello.py
Hello, David
> python hello.py
Hello, David
```

3.5 Guessing Game

Another frequent example of a simple program is a guessing game. It might looks something like this:

```python
import random
import sys

guesses = []
max = 50
name = input("Hello, what is your name? ")
number = random.randint(1, max)
print("{}, I've picked a number between 1 and {}.".format(name, max))

while len(guesses) < 6:
    print('Take a guess.')
    guess = int(input('> '))
    if guess in guesses:
        print("You already tried that number!")
        continue
    if guess < number:
        print('Your guess is too low.')
    if guess > number:
        print('Your guess is too high.')
    if guess == number:
        print('You guessed my number in {} tries, {}.'.format(
            len(guesses)+1, name))
        break
    guesses.append(guess)
else:
    print("Too many guesses, {}!"
          "  The number I was thinking of was {}".format(
                name, number))
```

A playing session might look like this:

```
Hello, what is your name? David
David, I've picked a number between 1 and 50.
Take a guess.
> 25
Your guess is too low.
Take a guess.
> 40
```

```
Your guess is too low.
Take a guess.
> 45
Your guess is too low.
Take a guess.
> 48
You guessed my number in 4 tries, David.
```

The magic of persistence is that everything is remembered between program runs. So lets rewrite this so instead of a loop, we're using commands typed at the shell prompt to play the game.

First, we need a command to start the game:

```
#!/usr/bin/env python

import os
import sys
import random
from nvm.pmemobj import create, PersistentList, PersistentDict

pool_fn = 'guessing_game.pmem'
max = 50

try:
    pool = create(pool_fn)
except OSError as err:
    print(err)
    print("Are you already in the middle of a game?")
    sys.exit()

with pool:
    with pool.transaction():
        root = pool.root = pool.new(PersistentDict)
        root['number'] = random.randint(1, max)
        root['name'] = name = input("Hello, what is your name?  ")
        root['guesses'] = pool.new(PersistentList)
print("{}, I've picked a number between 1 and {}.".format(name, max))
print("Type 'guess' followed by your guess at the prompt.")
```

This introduces several new concepts. The `create()` function raises an error if the persistent memory file already exists. This is equivalent to specifying `flag='x'` in the `PersistentObjectPool` constructor. This command is only dealing with creating the pool, so it doesn't have an if test to see if root is `None`, it can just go ahead and do the setup.

However, we want the setup to either work or fail completely, so we use the pool's `transaction()` context manager to wrap all of our initialization in a transaction.

The first thing we do is create a namespace to hold our persistent program data. We use a dictionary for this, but we can't persist a normal Python dict. Instead we use the `pmemobj.PersistentDict`. To create one, we use the `new()` method of the pool. The `new` method requires a class object that supports the `Persistent` interface, and given one it creates an instance of the object that will store its data in the pool. We could also pass constructor arguments after the class name. `PersistentDict` accepts the same constructor arguments as a normal dict.

Note that in addition to giving the `PersistentDict` a local name, we also assign it to the `root` attribute of the pool. If we failed to do that, `pmemobj` would forget about the object once the pool was closed, since nothing would be referring to it. That is, when the pool is closed, `pmemobj` looks through all the objects in the pool, and any that cannot be reached from `root` are garbage collected.

Once we have our namespace, we store the player's name, and the number we want them to guess, and create an empty

PersistentList in which to store the guesses.

Then we tell the player what to do next, and we're ready to play.

We've told the player to type `guess <their_guess>` at the command line, so now we need to implement the `guess` command:

```
#!/usr/bin/env python

import os
import sys
from nvm.pmemobj import open

pool_fn = 'guessing_game.pmem'

if len(sys.argv) != 2:
    print("Please specify a single integer as your guess.")
    sys.exit(1)
try:
    guess = int(sys.argv[1])
except ValueError as err:
    print("Please specify an integer as your guess.")
    sys.exit(1)

try:
    pool = open(pool_fn)
except OSError as err:
    print(err)
    print("Perhaps you need to run 'start_guessing' first?")
    sys.exit(1)

if pool.root is None:
    # The start_guessing script must have been killed before
    # initialization was complete.
    print("Looks like a start was aborted.  Please run"
          " start_guessing again.")
    pool.close()
    os.remove(pool_fn)
    sys.exit()

with pool:
    done = False
    root = pool.root
    guesses = root['guesses']
    name = root['name']
    number = root['number']
    if guess in pool.root['guesses']:
        print("You already tried {}".format(guess))
    elif guess < number:
        print("Your guess is too low.")
    elif guess > number:
        print("Your guess is too high.")
    elif guess == number:
        print('You guessed my number in {} tries, {}.'.format(
                len(guesses)+1, name))
        done = True
    guesses.append(guess)
    if not done and len(guesses) > 6:
        print("Too many guesses, {}!"
              "  The number I was thinking of was {}".format(
                    name, number))
          done = True
if done:
    os.remove(pool_fn)
```

Here we've used `open()` to get access to the existing pool. It will throw an error if the pool does *not* exist. This is equivalent to passing `flag='r'` to the constructor.

We do need to check `root` to see if it is `None` here, since it could be if the initialization did not complete. In that case we just delete the pool and tell the player to start over from the beginning.

Notice how we can use a local name for the `guess` list, and append to it, and the list is updated persistently. This is because each `Persistent` class knows which pool it belongs to, so it can find the persistent memory it needs to update.

The other thing to notice about this example is that we haven't used an explicit transaction anywhere. We only do one data structure update, and that's the append of the new guess to the list. That append is guaranteed to be atomic, so there is no need for an explicit transaction in this case.

With our persistent version of the guessing game, running the game looks like this:

```
> start_guessing
Hello, what is your name?  David
David, I've picked a number between 1 and 50.
Type 'guess' followed by your guess at the prompt.
> guess 25
Your guess is too low.
> guess 35
Your guess is too low.
> guess 45
Your guess is too high.
> guess 40
Your guess is too high.
> guess 38
Your guess is too high.
> guess 37
Your guess is too high.
> guess 36
You guessed my number in 7 tries, David.
```

Now, this code is somewhat more complicated than the non-persistent version, but it would allow you to start a game one day, and come back days later and finish the game. We could add a 'status' command that let you know now many guesses you'd made, and even replay the guesses. While this is a trivial example, I think you can see how these principles would apply to more useful programs with retained state.

3.6 Persistent Objects

Python is an object oriented language, so we of course would like to be able to persist arbitrary objects. We can't do that in the general case, since anything that has a specific memory layout requires specific support in *`pmemobj`*. However, Python objects that do not subclass built-in types are, from the point of view of persistent memory, just a dictionary wrapped in some extra behavior. So *`pmemobj`* does support persisting arbitrary objects that do not subclass built-ins, via the *`PersistentObject`* base class.

Our `guess` code above has to awkwardly pull the data of interest out of the dictionary that we used as a namespace. It would provide simpler code if we can instead have that data be attributes on an object. To do that, we'll need to be able to access that object from both programs, so we'll want a separate python file to hold our class definition:

```python
import random
import os

from nvm.pmemobj import open, PersistentObject, PersistentList

pool_fn = 'guessing_game2.pmem'


class GameError(Exception):
    pass

def reopen_game():
    if not os.path.isfile(pool_fn):
        raise GameError("No game in progress.  Use 'start_guessing'"
                        " to start one.")
    try:
        pool = open(pool_fn)
    except OSError as err:
        exc = GameError("Could not open game file: {}".format(err))
        try:
            os.remove(pool_fn)
        except OSError as err:
            raise GameError("Can't remove game file")
        raise GameError("Could not open game file, start again"
                        " with 'start_guessing'")
    if pool.root is None:
        pool.close()
        os.remove(pool_fn)
        raise("Looks like a game was aborted; start again with"
              " 'start_guessing'")
    return pool


class Guesser(PersistentObject):

    def __init__(self, name, maximum=50):
        self.name = name
        self.maximum = maximum
        self.number = random.randint(1, maximum)
        self.guesses = self._p_mm.new(PersistentList)
        self.lost = False
        self.done = False

    def _guess_to_int(self, s):
        try:
            guess = int(s)
        except ValueError as err:
            raise ValueError("Please specify an integer; {} is not"
                             "valid: {}".format(s, err))
        if guess < 1 or guess > self.maximum:
            raise ValueError("Come now, {}, a guess outside of the"
                             " range I told you won't get you"
                             " anywhere".format(self.name))
        return guess

    def check_guess(self, guess):
        guess = self._guess_to_int(guess)
        with self._p_mm.transaction():
            self.current_guess = guess
            if guess in self.guesses:
                self.current_outcome = 'SEEN'
            self.guesses.append(guess)
            if guess == self.number:
                self.current_outcome = 'EQUAL'
                self.done = True
            if len(self.guesses) > 6:
                self.lost = True
                self.done = True
            if guess < self.number:
                self.current_outcome = 'LOW'
            if guess > self.number:
                self.current_outcome = 'HIGH'
        return self.current_outcome

    def message(self, key):
        return getattr(self, 'msg_' + key)()

    def msg_START(self):
        return "{}, I've picked a number between 1 and {}.".format(
                    self.name, self.maximum)

    def msg_SEEN(self):
        return "You already tried {}".format(self.current_guess)

    def msg_EQUAL(self):
        return "You guessed my number in {} tries, {}.".format(
                    len(self.guesses), self.name)

    def msg_LOW(self):
        return "Your guess is too low."

    def msg_HIGH(self):
        return "Your guess is too high."

    def msg_LOST(self):
        return ("Too many guesses, {}!"
                "  The number I was thinking of was {}".format(
                    self.name, self.number))
```

The first thing to notice about the `PersistentObject` subclass is that for the most part it doesn't look any different from a normal Python class. There is an `__init__` that is executed when the object is first created, and most attributes are referenced and set normally. The one exception is our `self.guesses` attribute. We want that to be a list. Since it is not a non-container immutable, it needs to be a `Persistent` object itself.

To accomplish this we make use of the `_p_mm` attribute of our `PersistentObject` instance. This attribute points to the `MemoryManager` instance associated with the `PersistentObject`. We can use that reference to access the `MemoryManager's` `new()` method, and use that method to create an empty `PersistentList` that is associated with the same `Memorymanager` managing our `PersistentObject`.

We can also use the `_p_mm` attribute to access the `MemoryManager's` `transaction()` context manager, as you can see in the `check_guess` method of the example. Unlike our previous example, in this code block we are making several updates to our class that should either all be done, or none of them done. By using the transaction, we ensure that either the guess is completely processed, or it is not processed at all, no matter when the program gets interrupted.

With the game logic now factored out into a class, our command scripts are much simpler.

`start_guessing` becomes:

```
#!/usr/bin/env python

import os
import sys
from nvm.pmemobj import create
from guess_lib import Guesser, pool_fn

name = input("Hello, what is your name?  ")
if os.path.isfile(pool_fn):
    print("There is already a game file.  Use the guess_status command"
          " to see details of the current game.")
    sys.exit(1)
try:
    pool = create(pool_fn)
except OSError as err:
    print(err)
    sys.exit(err.errno)

with pool:
    pool.root = game = pool.new(Guesser, name)
    print(game.message('START'))
    print("Type 'guess' followed by your guess at the prompt.")
```

To start the game, we check there's no existing game file and create it, but now initializing the data structures in the pool consists of just calling *`new()`* on our `guesser` class and assigning that to `root`.

The `guess` command is now almost trivial:

```
#!/usr/bin/env python

import os
import sys
from guess_lib import reopen_game, GameError

if len(sys.argv) != 2:
    print("Please specify a single integer as your guess.")
    sys.exit(1)
guess = sys.argv[1]

try:
    pool = reopen_game()
except (OSError, GameError) as err:
    print(err)
    sys.exit(1)

with pool:
    guesser = pool.root
    try:
        disposition = guesser.check_guess(guess)
    except ValueError as err:
        print(err)
        sys.exit(1)
    print(guesser.message(disposition))
    if guesser.lost:
        print(guesser.message('LOST'))
if guesser.done:
    os.remove(pool_fn)
```

We use our library function to reopen the pool, which checks for the various error conditions and aborts with the appropriate message if we run into any of them. Then we grab the `guesser` instance from the pool's `root` and pass the guess the player made its `check_guess` method to evaluate, printing the message associated with whatever guess status it returns, removing the game file if and only if the game is over:

And now we can easily implement the `game_status` command mentioned earlier:

```python
#!/usr/bin/env python

import os
import sys

from guess_lib import reopen_game, GameError

try:
    pool = reopen_game()
except (OSError, GameError) as err:
    print(err)
    sys.exit(1)

with pool:
    guesser = pool.root
    if not guesser.guesses:
        print("No guesses yet, use 'guess <integer>' to make a guess")
        sys.exit(0)
    print("guesses so far:")
    for guess in guesser.guesses:
        print("  {}".guess)
    print("my response to your last guess:")
    print("  {}".guesser.message(guesser.current_outcome))
```

The pattern here is one I expect many persistent memory applications will share (possibly via a single program with subcommands or sub-functions, rather than the multiple program files in this example): the persistent memory is accessed through an instance of an application specific class that is assigned to the `root` of the object pool. When run, the application makes sure it can access the pool, then grabs the instance from `root` and uses the instance's methods to accomplish the application's goals.

CHAPTER 4

pmem – low level persistent memory support

See also:

PMDK libpmem man page.

class nvm.pmem.**DrainContext**(*memory_buffer*, *unmap=True*)
A context manager that will automatically drain the specified memory buffer.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**FILE_CREATE = 1**
Create the named file if it does not exist.

nvm.pmem.**FILE_EXCL = 2**
Ensure that this call creates the file.

nvm.pmem.**FILE_SPARSE = 4**
When creating a file, create a sparse (holey) file instead of calling posix_fallocate(2)

nvm.pmem.**FILE_TMPFILE = 8**
Create a mapping for an unnamed temporary file.

class nvm.pmem.**FlushContext**(*memory_buffer*, *unmap=True*)
A context manager that will automatically flush the specified memory buffer.

Parameters **memory_buffer** – the MemoryBuffer object.

class nvm.pmem.**MemoryBuffer**(*buffer_*, *is_pmem*, *mapped_len*)
A file-like I/O (similar to cStringIO) for persistent mmap'd regions.

read(*size=0*)
Read data from the buffer.

Parameters **size** – size to read, zero equals to entire buffer size.

Returns data read.

seek(*pos*)
Moves the cursor position in the buffer.

Parameters **pos** – the new cursor position

write(*data*)
Write data into the buffer.

Parameters **data** – data to write into the buffer.

nvm.pmem.**check_version**(*major_required*, *minor_required*)
Checks the libpmem version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

nvm.pmem.**drain**(*memory_buffer=None*)
Wait for any PM stores to drain from HW buffers.

Parameters **memory_buffer** – the MemoryBuffer object, legacy param, not required anymore.

nvm.pmem.**flush**(*memory_buffer*)
Flush processor cache for the given memory region.

Parameters **memory_buffer** – the MemoryBuffer object.

nvm.pmem.**has_hw_drain**()
This function returns true if the machine supports the hardware drain function for persistent memory, such as that provided by the PCOMMIT instruction on Intel processors.

Returns return True if it has hardware drain, False otherwise.

nvm.pmem.**is_pmem**(*memory_buffer*)
Return true if entire range is persistent memory.

Returns True if the entire range is persistent memory, False otherwise.

nvm.pmem.**map_file**(*file_name*, *file_size*, *flags*, *mode*)
Given a path, this function creates a new read/write mapping for the named file. It will map the file using mmap, but it also takes extra steps to make large page mappings more likely.

If creation flags are not supplied, then this function creates a mapping for an existing file. In such case, *file_size* should be zero. The entire file is mapped to memory; its length is used as the length of the mapping.

See also:

PMDK libpmem documentation.

Parameters

- **file_name** – The file name to use.
- **file_size** – the size to allocate
- **flags** – The flags argument can be 0 or bitwise OR of one or more of the following file creation flags: *FILE_CREATE*, *FILE_EXCL*, *FILE_TMPFILE*, *FILE_SPARSE*.

Returns The mapping, an exception will rise in case of error.

nvm.pmem.**msync**(*memory_buffer*)
Flush to persistence via *msync()*.

Parameters **memory_buffer** – the MemoryBuffer object.

Returns the msync() return result, in case of msync() error, an exception will rise.

`nvm.pmem.`**`persist`**`(`*memory_buffer*`)`

Make any cached changes to a range of pmem persistent.

Parameters **`memory_buffer`** – the MemoryBuffer object.

`nvm.pmem.`**`unmap`**`(`*memory_buffer*`)`

Unmap the specified region.

Parameters **`memory_buffer`** – the MemoryBuffer object.

CHAPTER 5

`pmemlog` – pmem-resident log file

See also:

PMDK libpmemlog man page.

class nvm.pmemlog.**LogPool**(*log_pool*)

This class represents the Log Pool opened or created using *create()* or *open()*.

append(*buf*)

This method appends from buffer to the current write offset in the log memory pool plp. Calling this function is analogous to appending to a file. The append is atomic and cannot be torn by a program failure or system crash.

On success, zero is returned. On error, -1 is returned and errno is set.

close()

This method closes the memory pool. The log memory pool itself lives on in the file that contains it and may be re-opened at a later time using *open()*.

nbyte()

This method returns the amount of usable space in the log pool. This method may be used to determine how much usable space is available after libpmemlog has added its metadata to the memory pool.

Note: You can also use *len()* to get the usable space.

Returns amount of usable space in the log pool.

rewind()

This method resets the current write point for the log to zero. After this call, the next append adds to the beginning of the log.

tell()

This method returns the current write point for the log, expressed as a byte offset into the usable log space in the memory pool. This offset starts off as zero on a newly-created log, and is incremented by each successful append operation. This function can be used to determine how much data is currently in the log.

Returns the current write point for the log, expressed as a byte offset.

walk (*func*, *chunk_size=0*)

This function walks through the log pool, from beginning to end, calling the callback function for each chunksize block of data found. The chunksize argument is useful for logs with fixed-length records and may be specified as 0 to cause a single call to the callback with the entire log contents.

Parameters

- **chunk_size** – chunk size or 0 for total length (default to 0).
- **func** – the callback function, should return 1 if it should continue walking through the log, or 0 to terminate the walk.

`nvm.pmemlog.`**check** (*filename*)

This method performs a consistency check of the file indicated and returns *True* if the memory pool is found to be consistent. Any inconsistencies found will cause this function to return False, in which case the use of the file with libpmemlog will result in undefined behavior.

Returns True if memory pool is consistent, False otherwise.

`nvm.pmemlog.`**check_version** (*major_required*, *minor_required*)

Checks the libpmemlog version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

`nvm.pmemlog.`**create** (*filename*, *pool_size=<Mock id='139648399243984'>*, *mode=438*)

The *create()* function creates a log memory pool with the given total *pool_size*. Since the transactional nature of a log memory pool requires some space overhead in the memory pool, the resulting available log size is less than poolsize, and is made available to the caller via the *nbyte()* function.

Note: If the error prevents any of the pool set files from being created, this function will raise an exception.

Parameters

- **filename** – specifies the name of the memory pool file to be created.
- **pool_size** – the size of the pool (defaults to PMEMLOG_MIN_POOL).
- **mode** – specifies the permissions to use when creating the file.

Returns the new log memory pool created.

Return type *LogPool*

`nvm.pmemlog.`**open** (*filename*)

This function opens an existing log memory pool, returning a memory pool.

Note: If an error prevents the pool from being opened, this function will rise an exception.

Parameters **`filename`** – Filename must be an existing file containing a log memory pool as created by the *`create()`* method. The application must have permission to open the file and memory map it with read/write permissions.

Returns the log memory pool.

Return type *LogPool*

CHAPTER 6

`pmemblk` – arrays of pmem-resident blocks

See also:

PMDK libpmemblk man page.

class `nvm.pmemblk.`**`BlockPool`**`(`*block_pool*`)`

This class represents the Block Pool opened or created using *`create()`* or *`open()`*.

`bsize``()`

This method returns the block size of the specified block memory pool. It's the value which was passed as block size to *`create()`*.

Returns the block size.

`close``()`

This method closes the memory pool. The block memory pool itself lives on in the file that contains it and may be re-opened at a later time using *`open()`*.

`nblock``()`

This method returns the usable space in the block memory pool, expressed as the number of blocks available.

Returns usable space in block memory pool in number of blocks.

`read``(`*block_num*`)`

This method reads a block from memory pool at specified block number.

Note: Reading a block that has never been written will return an empty buffer.

Returns data at block.

`set_error``(`*block_num*`)`

This method sets the error state for block number blockno in memory pool. A block in the error state returns errno EIO when read. Writing the block clears the error state and returns the block to normal use.

Returns On success, zero is returned. On error, an exception will be raised.

set_zero(*block_num*)
This method writes zeros to block number blockno in memory pool. Using this function is faster than actually writing a block of zeros since libpmemblk uses metadata to indicate the block should read back as zero.

Returns On success, zero is returned. On error, an exception will be raised.

write(*data*, *block_num*)
This method writes a block from data to block number blockno in the memory pool. The write is atomic with respect to other reads and writes. In addition, the write cannot be torn by program failure or system crash; on recovery the block is guaranteed to contain either the old data or the new data, never a mixture of both.

Returns On success, zero is returned. On error, an exception will be raised.

nvm.pmemblk.**check**(*filename*, *block_size=0*)
This function performs a consistency check of the file indicated by path and returns True if the memory pool is found to be consistent. Any inconsistencies found will cause it to return False, in which case the use of the file with libpmemblk will result in undefined behavior.

Note: When block size is non-zero, it will compare it to the block size of the pool and return False when they don't match.

Returns True if memory pool is consistent, False otherwise.

nvm.pmemblk.**check_version**(*major_required*, *minor_required*)
Checks the libpmemblk version according to the specified major and minor versions required.

Parameters

- **major_required** – Major version required.
- **minor_required** – Minor version required.

Returns returns True if the nvm has the required version, or raises a RuntimeError exception in case of failure.

nvm.pmemblk.**create**(*filename*, *block_size=<Mock id='139648400388048'>*, *pool_size=<Mock id='139648400175248'>*, *mode=438*)
This function function creates a block memory pool with the given total pool size divided up into as many elements of block size as will fit in the pool.

Note: Since the transactional nature of a block memory pool requires some space overhead in the memory pool, the resulting number of available blocks is less than poolsize / block size, and is made available to the caller via the *nblock()*.

If the error prevents any of the pool set files from being created, this function will raise an exception.

Parameters

- **filename** – specifies the name of the memory pool file to be created.
- **block_size** – the size of the blocks (defaults to PMEMBLK_MIN_BLK).
- **pool_size** – the size of the pool (defaults to PMEMBLK_MIN_POOL).
- **mode** – specifies the permissions to use when creating the file.

Returns the new block memory pool created.

Return type *BlockPool*

`nvm.pmemblk.`**`open`**(*filename*, *block_size=0*)

This function opens an existing block memory pool, returning a memory pool.

Note: If an error prevents the pool from being opened, this function will rise an exception. If the block size provided is non-zero, it will verify the given block size matches the block size used when the pool was created. Otherwise, it will open the pool without verification of the block size.

Parameters **`filename`** – Filename must be an existing file containing a block memory pool as created by the `create()` method. The application must have permission to open the file and memory map it with read/write permissions.

Returns the block memory pool.

Return type *BlockPool*

CHAPTER 7

pmemobj — Pesistent Python objects

7.1 Creating and Accessing a PersistentObjectPool

pmemobj.**create**(*filename*, *pool_size=MIN_POOL_SIZE*, *mode=0o666*, *debug=False*)

Return a *PersistentObjectPool* backed by a file named *filename*, allocating *pool_size* bytes for the pool, and setting the mode of the file on the filesystem to *mode*. Raise an `OSError` if the file already exists. Pass *debug* to the *PersistentObjectPool* constructor.

If *filename* is in a filesystem backed by persistent memory, the memory will be directly accessed. Otherwise persistent memory will be emulated by memory mapping a disk file.

The actual amount of memory available for objects is smaller than *pool_size* transaction and object management overhead. The default is the default used by `libpmemobj`.

pmemobj.**open**(*filename*, *debug=False*)

Return a *PersistentObjectPool* backed by the file named *filename*. Raise an an `OSError` if the file does not exist. If the previous shutdown was not clean, call the *PersistentObjectPool.gc* method. Pass *debug* to the *PersistentObjectPool* constructor.

class pmemobj.**PersistentObjectPool**(*filename*, *flag='w'*, *pool_size=MIN_POOL_SIZE*, *mode=0x666*, *debug=False*)

Open or create a persistent object pool backd by *filename*. If *flag* is `w`, raise an `OSError` if the file does not exist and otherwise open it for reading and writing. If *flag* is `x`, raise an `OSError` if the file already exists, and otherwise create the file and open it for reading and writing. If *flag* is `c`, create the file if it does not exist, but in any case open it for reading and writing.

If the file gets created, allocate *pool_size* bytes for the pool, and set its mode in the filesystem to *mode*.

If the object pool was previously not closed cleanly, call *gc()*.

Use *debug* as the default value for the *debug* parameter to the *gc()* method.

root

The "root" object of the pool. This can be set to any object that can be persisted, but it is really only useful to set it to a Perisistent collection type. Only objects that are reachable by traversing the object graph starting from the root object will be preserved once the object pool is closed.

gc(*debug=None*)
Free all unreferenced objects: objects not accessible by tracing the object graph starting at the `root` object.

new(*typ*, **args*, ***kw*)
Create a new instance of *typ* managed by this pool, passing its constructor *args* and *kw*. *typ* must support the `Persistent` API.

persist_via_pickle(**types*)
Add *types* to the list of types that will be persisted via pickle. Nominated types must be non-container immutable types (this is not currently enforced, but confusing things will happen if you violate it). If a version of pmemobj to which support for a given type has been added is used to open a pool with instances of that type stored via pickle, the object will be resurrected from pickle, but any new instances written to the pool will use the direct support.

transaction()
Return a context manager object that manages a transaction. If the context is exited normally, all changes to objects managed by the pool should be committed; if the context exits abnormally or the program stops running for any reason in the middle of the context, then none of the changes to the persistent objects inside the transaction context should be visible. Note that the transaction does not affect changes to normal Python objects; only changes to Persistent objects will be rolled back on abnormal exit.

close()
Call `gc()`, mark the pool as clean, and close the underlying file. The object pool lives on in the file that contains it and may be reopened at a later date, and all the objects reachable from the `root` object will be in the same state they were in when the pool was closed.

7.2 Managing Persistent Memory

class pmemobj.**MemoryManager**(*pool_pr*, *type_table=None*)
Create a manager for a `PersistentObjectPool`'s memory. This class should never be instantiated directly, but instead the automatically created instance should be accessed through a pool object.

All of the methods below are atomic from the point of view of the caller. If the program crashes in the middle of the method it will either have completed or on pool reopen it will be as if it had never been started. All methods may be called from inside a transaction to make them part of a larger atomic unit of change.

new(*typ*, **args*, ***kw*)
Create a new instance of *typ* managed by the pool, passing its constructor *args* and *kw*. *typ* must support the `Persistent` API.

transaction()
Return a context manager object that manages a transaction. If the context is exited normally, all changes to objects managed by the pool should be committed; if the context exits abnormally or the program stops running for any reason in the middle of the context, then none of the changes to the persistent objects inside the transaction context should be visible when the pool is next opened.

otuple(*oid*)
Ensure that *oid* is in tuple form. An `oid` retreived from memory is actually a pointer to the memory the oid was retrieved from, so if contents of that memory location changes the value of the raw `oid` would change. This method copies the `oid` data into a tuple not subject to such modification, but which can be assigned to a memory field to store its value at that location.

All `MemoryManager` methods that return oids return them in tuple form.

alloc(*size*, *type_num=POBJECT_TYPE_NUM*)
Return an `oid` pointing to *size* bytes of newly allocated persistent memory, passing *type_num* to libpmemobj as the new memory object's type. Raise an error if called outside of any `transaction()`.

A *Persistent* class should use POBJECT_TYPE_NUM for its base memory allocation, but should use a unique number for any non-PObject memory structures it allocates. (There is currently no way to manage allocating these numbers to guarnatee uniqueness, but in fact as long as something other than POBJECT_TYPE_NUM is used, nothing should break even if the number collides with at used by a different *Persistent* type, you just lose some memory type safety.)

zalloc(*size*, *type_num=POBJECT_TYPE_NUM*)
Same as *alloc()*, but the allocated persistent memory is also zeroed.

free(*oid*)
Return the persistent memory pointed to by *oid* to the pool, so that it is avaiable for future allocation. Raise an error if called outside of any transaction.

realloc(*oid*, *size*, *type_num=None*)
Return an `oid` pointing to *size* bytes of newly allocated persistent memory and copy the data pointed to by *oid* into it, truncating or zero-filling as needed. Raise an error if *type_num* is not `None` and does not match the pmem type of *oid*. *free()* the memory originally pointed to by *oid*. Raise an error if called outside of any transaction.

zrealloc(*size*, *type_num=POBJECT_TYPE_NUM*)
Same as *realloc()*, but the newly allocated persistent memory is also zeroed.

incref(*oid*)
Increment the reference count of the `PObject` pointed to by *oid*.

decref(*oid*)
Decrement the reference count of the `POjbect` pointed to by *oid*. If the reference count is zero after the decrement, then if the object has a `_p_deallocate()` method call it, and in any case call *free()* on *oid*.

xdecref(*oid*)
Call *decref()* on *oid* if *oid* is not `OID_NULL`.

decref() should be used whever possible, so that cases where an `oid` is unexpectedly null raise an error. If, however, the poitner can legitimately be null, this method eliminates the need for an if test, and this is a common enough case to be worth having extra method.

persist(*obj*)
Return an `oid` pointing to the representation of *obj* in peristent memory, creating that representation if necessary. *obj* must be one of the directly supported immutable types, or one of the immutable types nominated for persistence via `pickle`, or a *Persistent* type.

resurrect(*oid*)
Return a Python object representing the `POjbect` stored at *oid*. This may be a pure Python object if the stored object is a non-container immutable, or is otherwise an object that redirects data accesses to data stored in persistent memory.

direct(*oid*)
Return the real memory address of the persistent memory pointed to by *oid*.

7.3 Persistent Classes

class pmemobj.**Persistent**
Persistent is an Abstract Base Class for objects that implement the `Persistent` interface. All classes that want to store their data in persistent memory and manage it must implement the interface described here, but they are not required to use the ABC as their base.

`_p_mm`
A `MemoryManager` instance from the `PersistentObjectPool` in which this object is stored.

`_p_oid`
The `oid` that points to the `PObject` data structure in persistent memory that anchors this objects data.

`_p_new`(*manager*)
Initialize the objects data structures when the object is initially created, and store the provided `MemoryManager` *manager* in `_p_mm` and the `oid` pointing to the initialized data structures (a `POjbect`) in `_p_oid`.

`_p_resurrect`(*manager*, *oid*)
Restore the object's state from the data located at *oid*, using *manager*, storing the *manager* in `_p_mm` and the *oid* in `_p_oid`.

`_p_traverse`()
Return an iterable over the `oids` of all of the objects pointed to by this object.

`_p_substructures`()
Return an iterable over the oids of all of the non-`PObject` data structures allocated by this object.

`_p_deallocate`()
Remove all pointers to any other objects, and `free()` any allocated data structures. When this method returns, only the memory pointed to by `_p_oid` should remain allocated.

class `pmemobj.`**`PersistentList`**([*iterable*])
A `Persistent` version of the normal Python `list`. Its behavior should be identical except for being persistent. (Note: currently slices are not supported.)

class `pmemobj.`**`PersistentDict`**([*mapping_or_iterable*], ***kwarg*)
A `Persistent` version of the normal Python `dict`. Its behavior should be identical except for being persistent.

class `pmemobj.`**`PersistentObject`**
Base class for user defined `Persistent` objects. May not be mixed with any other `Persistent` type.

As with a normal class, `__init__` is called when the object is initially created. It is *not* called during object resurrection.

`_v__init__`()

This method is called both when the object is initially created *and* when the object is resurrected. It does nothing by default, but can be overridden to (re)acquire volatile resources. It is called before `__init__` during object creation.

CHAPTER 8

What's new?

8.1 Version v.0.4

Changes in this version:

8.2 Version v.0.3

Changes in this version:

- Support for Python3.
- Initial pmemobj-based object manager with support for Persistent versions of several common Python mutable types (dictionary, list, set, tuple), specific support for various immutable types, and generic support for arbitrary non-container immutable types.
- Expanded documentation.

8.3 Version v.0.2

Changes in this version:

- Changes to mirror the nvml API changes regarding the pmem_map_file().
- Tests updates.

CHAPTER 9

License

Note: This library links with PMDK, you can see the PMDK license here. The pmemobj Python code is covered by the PMDK license, rather than the above.

Note: This framework is in active development and it is still in beta release.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index