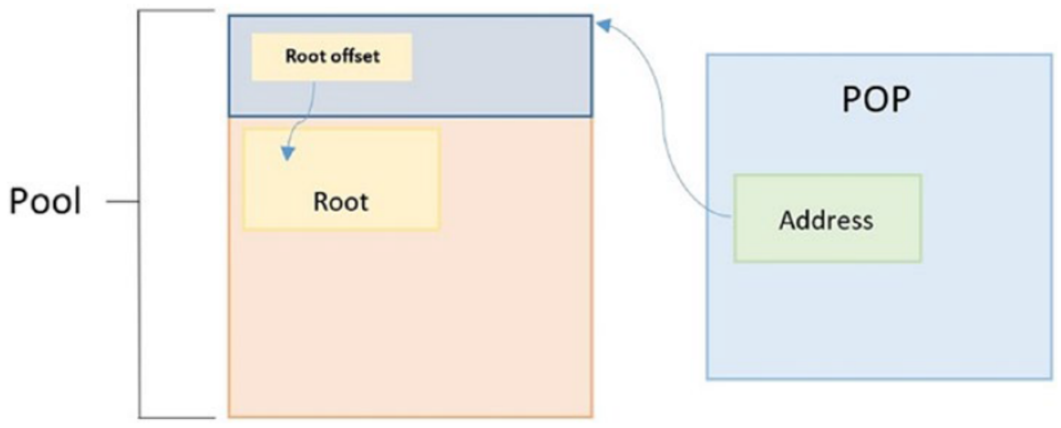
Persistent Memory Documentation

Libpmemobj

Root Object



The root object is what is always accessible from a given persistent memory pool. When constructing data structures, everything should originate from the root object and then have pointers to different regions of persistent memory. If you only intend to use a non-dynamically sized data structure, then you can solely use the root object. An example is shown below:

```
struct my_root {
  size_t len;
  char buf[MAX_BUF_LEN];
};
```

Otherwise it is recommended to allocate persistent memory and store the pointers within the root object. Each persistent object pointer is stored as a PMEMoid struct. This struct contains the id of a pool and the offset from the root object.

Traditional pointers such as char*, double*, etc. will not be sufficient. This is because the memory mappings of the persistent memory pool to the application address space will be different every time a program executes. Therefore the char* that points to persistent memory in the previous execution will not be valid when attempting to access these persistent memory regions upon later executions.
Every time you want to have a pointer to persistent memory, use a PMEMoid struct object. When accessing the pointers, you can then “cast” them to the corresponding correct pointers they correspond to with the pmemobj_direct() function. More details about the pmemobj_direct() function are explained later.

Here is an example of a struct that would be in normal memory and translating it to use for persistent memory:

DRAM

```
struct example{
        size_t buf_len;
        char* buf;
        double* num_ptr;
};
```

Persistent Memory

```
struct pmem_example{
        size_t buf_len;
        PMEMoid buf;
        PMEMoid  num_ptr;
};
```

To be able to access the Persistent Memory pointed regions, the pmemobj_direct() function can be called.pmemobj_direct takes in a PMEMoid struct and returns a void pointer to the mapped location in persistent memory. Based on the PMEMoid, cast the pointer to the corresponding pointer such as char*, double*, etc. Examples are shown below:

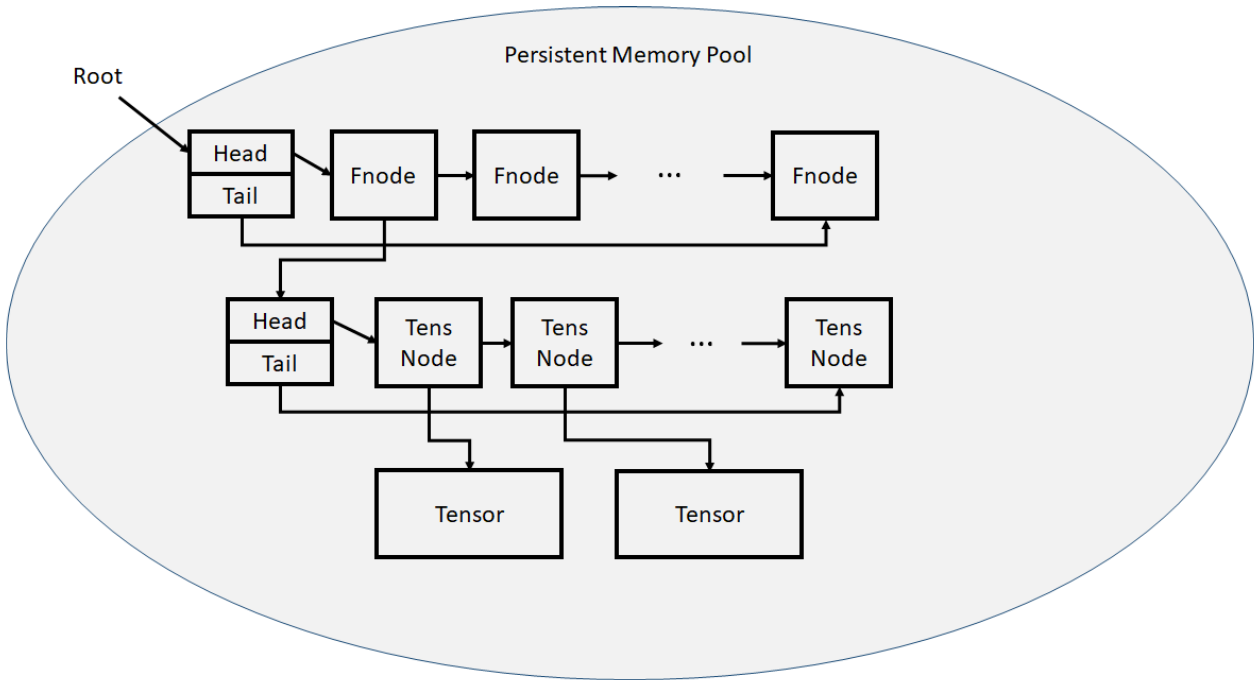
```
char* mapped_buf = pmemobj_direct(example->buf);
double* mapped_num_ptr = pmemobj_direct(example->num_ptr);
```

More information about libpmemobj can be found here:
https://pmem.io/pmdk/manpages/linux/v1.0/libpmemobj.3.html

To compile c programs with persistent memory use the following command:
```
gcc pool_pmem.c -L/lib/x86_64-linux-gnu/ -lpmemobj -o pool_pmem
```

Change /lib/x86_64-linux-gnu/ to the corresponding location of the libpmem.so files.

The main program for copying tensors to persistent memory can be found in pool_pmem.c.
Here is a brief explanation of how pool_pmem.c works:



Given a single persistent memory pool, files are represented as appendable linked lists where each node represents a given tensor added. The first link list contains nodes for each file. Each file is determined by its filename which is passed in when persisting a tensor. For a given file, tensor nodes are then appended to the tensor_list corresponding to a given file node. Each tensor node then points to a tensor in persistent memory which stores the raw data of the tensor.
How the persist_tensor function is defined, it treats a tensor as a string indicated by the event_str. This is to integrate with the Amazon Sagemaker Debugger which stores the raw tensor data within a google protocol buffer. Normally for checkpointing, it will serialize the tensor and its metadata into a byte string which is written. This will take the event string and write it to persistent memory.
Nodes as well as tensors are allocated using the pmemobj_zalloc() function. This will allocate persistent memory to a given PMEMoid variable where data can then be written. The persistent memory then needs to be persisted otherwise it will be lost once the program terminates. pmemobj_persist() and pmemobj_memcpy_persist() can be used to persist the persistent memory.

Amazon Sagemaker Debugger
Modifying the sagemaker debugger, the original source code can be found here:
sagemaker-debugger/smdebug at 7c5c8c4f1379c826e4ec2071ef5ecef91aa85b1d · awslabs/sagemaker-debugger (github.com)

After installing sagemaker debugger via pip install smdebug, the following file was modified:
~/.local/lib/python3.8/site-packages/smdebug/core/access_layer/file.py

The exact location may differ depending on your environment. However you will need to modify this file after installing smdebug. The provided file.py shown in the github shows how the file.py was modified to integrate with the persistent memory c programs. Ctypes was used so a given .so file must be created for ctypes to access. More information on ctypes can be found here: ctypes — A foreign function library for Python — Python 3.9.6 documentation

An .so file can be created with the following command:

```
gcc -fPIC pool_pmem.c -L/lib/x86_64-linux-gnu/ -lpmemobj -shared -o pool_pmem.so
```

--

Errors that may be encountered

--

Below are errors I encountered when working with persistent memory. There may be more errors that may be unaccounted for but these can be a launching pad to consider.

If you encounter a segmentation fault when opening the persistent memory pool make sure the path name in addition to the layout name is correct. If these are correct the error may occur that the persistent memory pool was not closed before attempting to open it again.

For pointers involving persistent memory, make sure to use PMEMoid. While using pointers such as char* will work initially, they are only active as long as the program is alive. If trying to access those same pointers, memory will be mapped differently to the addresses in persistent memory. Therefore once again use PMEMoid for whenever you are using a pointer to persistent memory.

It is important to consider the maximum size that can be passed into persistent memory. There was a realization that the segmentation fault and allocation error was caused due to the size of the memory pool. If the size of the memory pool is too small and persistent memory is allocated to exceed the size of the pool, a segmentation fault will occur.

When integrating between python and c, before passing strings to corresponding c functions, make sure the strings are byte strings. While the main code only involved reading strings from python, if you intend to modify strings that originated in python be sure to use create_string_buffer from ctypes. Ctypes documentation can provide more information on creating a mutable string buffer:
ctypes — A foreign function library for Python — Python 3.9.6 documentation