论文阅读与前期工作总结

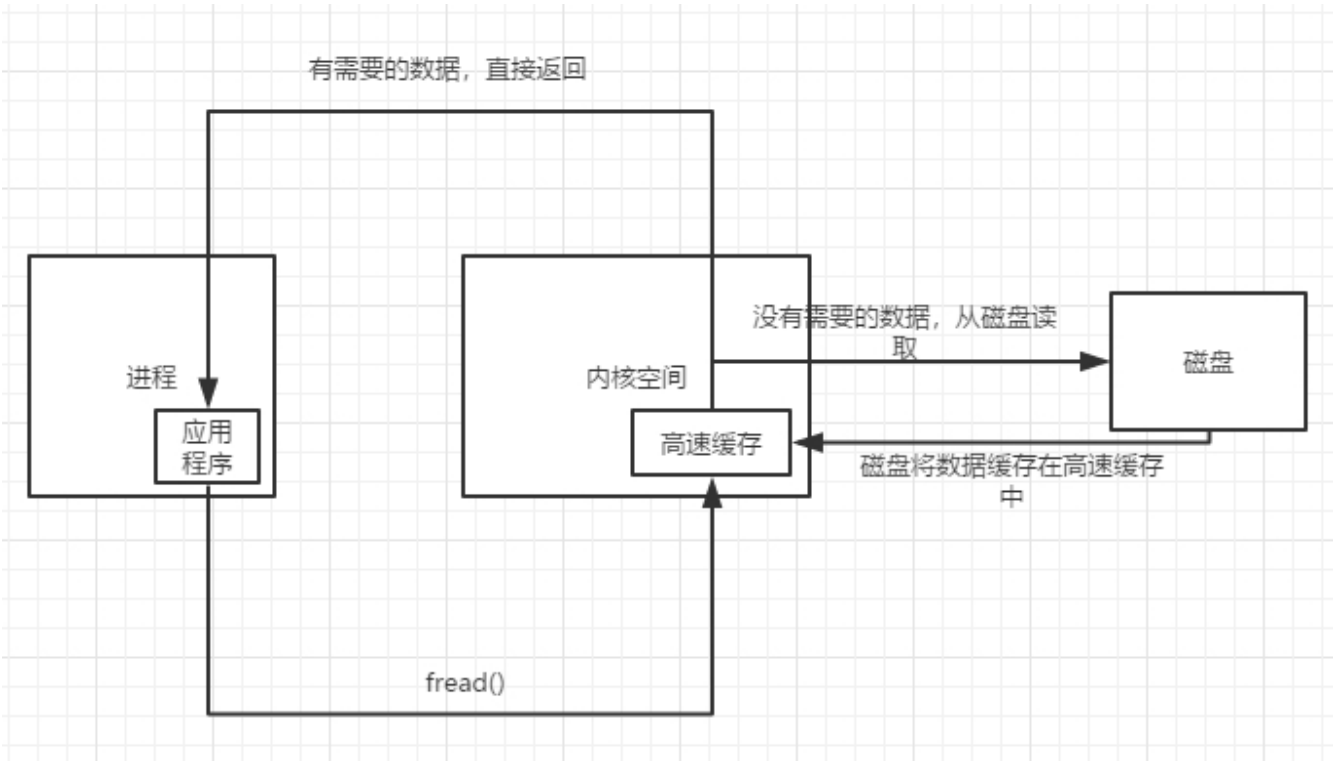
姓名：孔伟，黎福鑫，雷旭嘉，张莉斌

学号：17343055,17343058,17343057,16340290

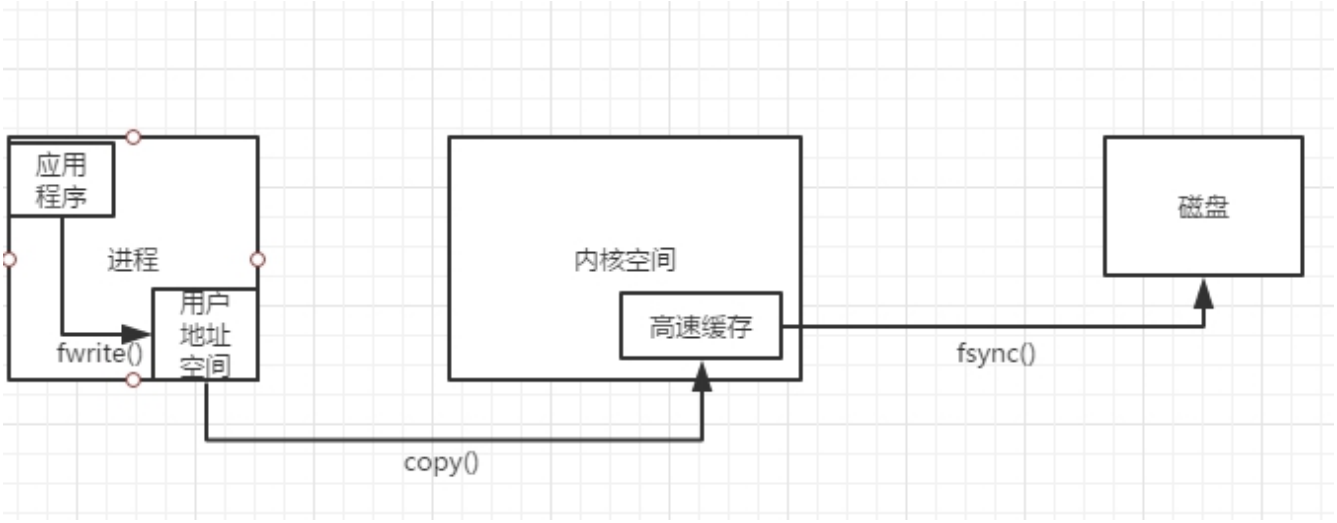
前期工作

1，使用示意图展示普通文件IO方式(fwrite等)的流程，即进程与系统内核，磁盘之间的数据交换如何进行？为什么写入完成后要调用fsync？

当应用程序调用read接口时，操作系统检查在内核的高速缓存有没有需要的数据，如果已经缓存了，那么就直接从缓存中返回，如果没有，则从磁盘中读取，然后缓存在操作系统的缓存中。



应用程序调用write接口时，将数据从用户地址空间复制到内核地址空间的缓存中，这时对用户程序来说，写操作已经完成，至于什么时候再写到磁盘中，由操作系统决定，除非显示调用了sync/fsync同步命令。



fsync函数：

功能：同步内存中所有已修改的文件数据到储存设备

过程：与sync不同，fsync()函数对指定文件描述符的单一文件生效，强制与该文件相连的所有修改过的数据传送到磁盘上，并且等待磁盘IO完毕，然后返回。当该函数返回0时，才真正表示成功保存到磁盘。数据库会在调用了write()之后调用fsync()。，

2，简述文件映射的方式如何操作文件。与普通IO区别？为什么写入完成后要调用msync？文件内容什么时候被载入内存?

Linux内核提供一种访问磁盘文件的特殊方式，将内存中某块地址空间和指定的磁盘文件相关联，从而将对该块内存的访问转换为对磁盘文件的访问。这种方式的目的是减少了数据从内核空间缓存到用户空间缓存的数据复制操作，当大量数据需要传输的时候，采用内存映射方式访问效率较高。

内存映射的步骤如下：用open系统调用打开文件, 并返回描述符fd；用mmap建立内存映射, 并返回映射首地址指针start；对映射(文件)进行各种操作, 显示(printf), 修改(sprintf)；用munmap(void *start, size_t lenght)关闭内存映射；用close系统调用关闭文件fd。

使用内存映射文件处理存储于磁盘上的文件时，将不必再对文件执行I/O操作，这意味着对文件进行处理时将不必再为文件申请并分配缓存，所有的文件缓存操作均由系统直接管理，由于取消了将文件数据加载到内存、数据从内存到文件的会写以及释放内存块等步骤，使得内存映射文件在处理大数据量的文件时能起到相当重要的作用。

在写入完成后，调用msync()，使磁盘上的文件内容与共享内存区的内容一致。

在内存映射过程中,并没有实际的数据拷贝,文件没有被载入内存,只是逻辑上放入了内存,具体到代码,就是建立并初始化了相关的数据结构,这个过程由系统调用mmap()实现,所以映射的效率很高.mmap函数将一个文件或者其他对象映射进内存，munmap执行相反的操作，删除特定地址区域的对象映射。成功执行时，mmap()返回被映射区的指针，munmap()返回0。失败时，mmap()返回MAP_FAILED[其值为(void *)-1]，munmap返回-1。

3，参考Intel的NVM模拟教程模拟NVM环境，用fio等工具测试模拟NVM的性能并与磁盘对比（关键步骤结果截图）。

（推荐Ubuntu 18.04LTS下配置，跳过内核配置，编译和安装步骤）

第一步，进行GRUB的配置。

以编辑模式打开grub文件：

```
kongwei@kongwei-VirtualBox:~$ sudo gedit /etc/default/grub
```

做出如下修改并保存：

```
GRUB_CMDLINE_LINUX=" memmap = 1G!4G"
```

第二步，sudo update-grub并重启后，可以看出，persistent已经建好。

```
雷旭嘉$>dmesg | grep user
[    0.000000] e820: user-defined physical RAM map:
[    0.000000] user: [mem 0x0000000000000000-0x000000000009fbff] usable
[    0.000000] user: [mem 0x000000000009fc00-0x000000000009ffff] reserved
[    0.000000] user: [mem 0x00000000000dc000-0x00000000000fffff] reserved
[    0.000000] user: [mem 0x0000000000100000-0x000000003ffebfff] usable
[    0.000000] user: [mem 0x000000003ffec000-0x000000003ffedfff] reserved
[    0.000000] user: [mem 0x000000003ffee000-0x000000003fffafff] ACPI data
[    0.000000] user: [mem 0x000000003fffb000-0x000000003fffffff] ACPI NVS
[    0.000000] user: [mem 0x00000000fd000000-0x00000000fd7fffff] reserved
[    0.000000] user: [mem 0x00000000fec00000-0x00000000fec00fff] reserved
[    0.000000] user: [mem 0x00000000fed00000-0x00000000fed00fff] reserved
[    0.000000] user: [mem 0x00000000fee00000-0x00000000fee00fff] reserved
[    0.000000] user: [mem 0x00000000ffc00000-0x00000000ffffffff] reserved
[    0.000000] user: [mem 0x0000000100000000-0x000000013fffffff] persistent (type 12)
[    4.850903] ppdev: user-space parallel port driver
```

第三步，安装fio。

```
雷旭嘉$>sudo make install
install -m 755 -d /usr/local/bin
install fio t/fio-genzipf t/fio-btrace2fio t/fio-dedupe tools/fio_generate_plots tools/plot/fio2gnuplot tools/genfio /usr/local/bin
install -m 755 -d /usr/local/man/man1
install -m 644 fio.1 /usr/local/man/man1
install -m 644 tools/fio_generate_plots.1 /usr/local/man/man1
install -m 644 tools/plot/fio2gnuplot.1 /usr/local/man/man1
install -m 755 -d /usr/local/share/fio
install -m 644 tools/plot/*gpm /usr/local/share/fio/
```

第四步，在硬盘上用fio测试性能：

```
kongwei@kongwei-VirtualBox:~/fio-2.2.5$ fio -filename=/tmp/test_randread -direct
=1 -iodepth 1 -thread -rw=randread -ioengine=psync -bs=16k -size=2G -numjobs=10
-runtime=60 -group_reporting -name=mytest
```

```
mytest: (groupid=0, jobs=10): err= 0: pid=7006: Tue Apr 23 00:48:52 2019
  read : io=145536KB, bw=2421.3KB/s, iops=151, runt= 60109msec
    clat (usec): min=110, max=585414, avg=65993.15, stdev=57646.05
     lat (usec): min=111, max=585416, avg=65994.05, stdev=57646.02
    clat percentiles (msec):
     |  1.00th=[   12],  5.00th=[   16], 10.00th=[   18], 20.00th=[   23],
     | 30.00th=[   29], 40.00th=[   35], 50.00th=[   45], 60.00th=[   58],
     | 70.00th=[   77], 80.00th=[  105], 90.00th=[  147], 95.00th=[  182],
     | 99.00th=[  265], 99.50th=[  306], 99.90th=[  441], 99.95th=[  482],
     | 99.99th=[  586]
    bw (KB  /s): min=   53, max=  533, per=10.10%, avg=244.63, stdev=78.65
    lat (usec) : 250=0.09%, 500=0.20%, 1000=0.01%
    lat (msec) : 4=0.03%, 10=0.29%, 20=13.93%, 50=40.27%, 100=23.74%
    lat (msec) : 250=20.16%, 500=1.25%, 750=0.03%
  cpu          : usr=0.04%, sys=0.20%, ctx=9128, majf=1, minf=51
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued    : total=r=9096/w=0/d=0, short=r=0/w=0/d=0, drop=r=0/w=0/d=0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: io=145536KB, aggrb=2421KB/s, minb=2421KB/s, maxb=2421KB/s, mint=60109mse
c, maxt=60109msec

Disk stats (read/write):
  sda: ios=9220/29, merge=76/31, ticks=608968/796, in_queue=610192, util=100.00%
```

其中重要信息：bw=2421.3kb/s,iops=151

第五步，创建文件系统并查看：

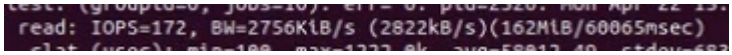
```
雷旭嘉$>sudo mkfs -t ext2 /dev/pmem0
mke2fs 1.42.13 (17-May-2015)
Creating filesystem with 262144 4k blocks and 65536 inodes
Filesystem UUID: f6c18af0-dc74-4b2a-816d-6f5f17a0d000
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Allocating group tables: 完成
正在写入inode表: 完成
Writing superblocks and filesystem accounting information: 完成

雷旭嘉$>sudo fdisk -l
Disk /dev/sda: 64 GiB, 68719476736 bytes, 134217728 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
Disklabel type: dos
Disk identifier: 0x7758973f

设备       启动      Start     末尾     扇区  Size Id 类型
/dev/sda1  *          2048 132120575 132118528   63G 83 Linux
/dev/sda2        132122622 134215679   2093058 1022M  5 扩展
/dev/sda5        132122624 134215679   2093056 1022M 82 Linux 交换 / Solaris

Partition 2 does not start on physical sector boundary.



Disk /dev/pmem0: 1 GiB, 1073741824 bytes, 2097152 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 4096 bytes
I/O size (minimum/optimal): 4096 bytes / 4096 bytes
```

创建挂载区域，mk /dev/usb1.

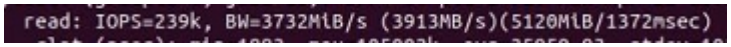
使用mount /dev/pmem0 /dev/usb1挂载后进行fio测试性能：

(由于内存过小，我问了实验成功的同学设置的虚拟机内存为6G,而问过组员电脑达不到这个程度，因此借用了那位同学的图进行观察)

硬盘测试：



NVM测试：



可以NVM性能高于硬盘性能。

关于以上截图用户名不一致的问题，因为实验是在组内成员多个电脑上同时进行，所以会出现一些不同的问题但是步骤都是一致的。

4，使用PMDK的libpmem库编写样例程序操作模拟NVM（关键实验结果截图，附上编译命令和简单样例程序）。

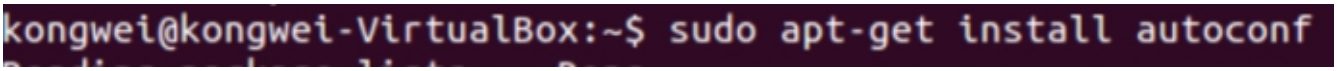
（样例程序使用教程的即可，主要工作是编译安装并链接PMDK库）

本题中要求的是使用PMDK的libpmen库编写杨丽程序操作模拟NVM，我首先编译了样例程序发现缺少<libpmen.h>那么这道题的理想状态就是安装好PMDK并且链接上后最终运行成功样例程序。

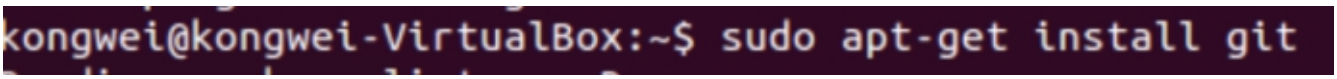
整个实验操作依据这篇简书博客的进行操作：

https://www.jianshu.com/p/bba1cdf01647

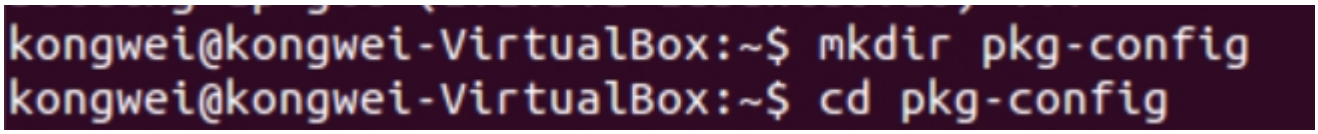
首先安装依赖包：



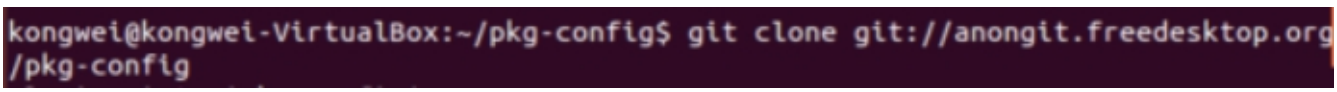
需要安装git进行后续的操作：



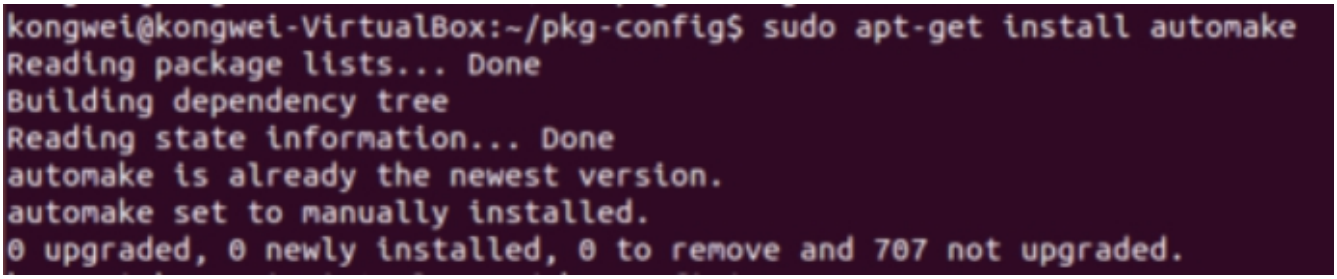
新建一个文件夹pkg-config并进入：



下载pkg-config:



安装automake：



增加一个参数：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ bash ./autogen.sh --with-int
ernal-glib
```

安装libtool：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo apt-get install libtool
```

执行./autogen.sh -- with-internal-glib

```
config.status: creating Makefile
config.status: creating glib/Makefile
config.status: creating glib/libcharset/Makefile
config.status: creating glib/gnulib/Makefile
config.status: creating m4macros/Makefile
config.status: creating config.h
config.status: executing depfiles commands
config.status: executing libtool commands
config.status: executing glib/glibconfig.h commands

Now type 'make' to compile pkg-config.
```

然后编译检查安装：（这里的命令一下子就被刷上去了）

make&&make check&&make install

下载ndctl安装包：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ git clone https://github.com/
pmem/ndctl.git
```

安装asciidoc：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo apt-get install asciidoc
```

安装xmlto:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo apt-get install xmlto
```

安装libkmod并配置:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ bash ./configure CFLAGS="-g -
02" --prefix=/usr --sysconfdir=/etc --libdir=/usr/lib
```

路径配置：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ export PKG_CONFIG_PATH=/usr/l
ib64/pkgconfig/:$PKG_CONFIG_PATH
```

安装uuid：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo apt-get install uuid-dev
el
```

安装json-c：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo apt-get install json-c-d
evel
```

make:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ make
make  all-recursive
make[1]: Entering directory `/home/kongwei/pkg-config/pkg-config'
Making all in glib
make[2]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
make  all-recursive
make[3]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
Making all in .
make[4]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
make[4]: Leaving directory `/home/kongwei/pkg-config/pkg-config/glib'
Making all in m4macros
make[4]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib/m4macros'
make  all-am
```

make check:

```
==================
All 30 tests passed
==================
make[2]: Leaving directory `/home/kongwei/pkg-config/pkg-config/check'
make[1]: Leaving directory `/home/kongwei/pkg-config/pkg-config/check'
```

make clean&&make&&makecheck:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ make clean&&make&&makecheck
Making clean in glib
make[1]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
Making clean in .
make[2]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
test -z "  " || rm -f
test -z "*.log *.trs " || rm -f *.log *.trs
test -z "" || rm -f
rm -rf .libs _libs
test -z "" || rm -f
```

make install:

存在这些错误：

```
make[2]: *** [install-binPROGRAMS] Error 1
make[2]: Leaving directory `/home/kongwei/pkg-config/pkg-config'
make[1]: *** [install-am] Error 2
make[1]: Leaving directory `/home/kongwei/pkg-config/pkg-config'
make: *** [install-recursive] Error 1
```

还是给个权限吧：

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo make install
```

之后没有出现错误了。

```
Making install in check
make[1]: Entering directory `/home/kongwei/pkg-config/pkg-config/check'
make[2]: Entering directory `/home/kongwei/pkg-config/pkg-config/check'
make[2]: Nothing to be done for `install-exec-am'.
make[2]: Nothing to be done for `install-data-am'.
make[2]: Leaving directory `/home/kongwei/pkg-config/pkg-config/check'
make[1]: Leaving directory `/home/kongwei/pkg-config/pkg-config/check'
```

至此，应该可以说依赖包安装完成。

开始正式安装PMDK:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ git clone https://github.com/
pmem/pmdk.git
```

make:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ make
make  all-recursive
make[1]: Entering directory `/home/kongwei/pkg-config/pkg-config'
Making all in glib
make[2]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
make  all-recursive
make[3]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
Making all in .
make[4]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib'
make[4]: Leaving directory `/home/kongwei/pkg-config/pkg-config/glib'
Making all in m4macros
make[4]: Entering directory `/home/kongwei/pkg-config/pkg-config/glib/m4macros'
make  all-am
```

sudo make install:

```
kongwei@kongwei-VirtualBox:~/pkg-config/pkg-config$ sudo make install
```

这样好就安装完了昨天一直苦苦尝试的PMDK。

下面进行libpmem库的安装：

```
li@li-virtual-machine:~$ sudo apt-get install libpmem-dev
```

full_copy.c程序的含义是，先建立一个文件(from.txt)作当源文件，当输入命令./a.out from.txt to.txt，该程序就会自动建立一个to.txt，并且将from.txt的内容完全复制到to.txt。

对full_copy.c进行编译：

```
li@li-virtual-machine:~/Desktop$ gcc full_copy.c -lpmem -o
a.out
li@li-virtual-machine:~/Desktop$
```

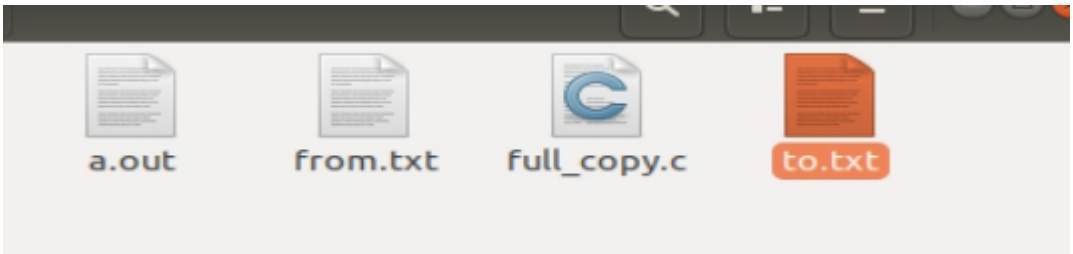
初始from.txt文件：

使用指令：

```
li@li-virtual-machine:~/Desktop$ ./a.out from.txt to.txt
li@li-virtual-machine:~/Desktop$
```

生成的to.txt文件：

最终文件条目：



论文阅读

总结一下本文的主要贡献和观点(500字以内)(不能翻译摘要)。

（回答本文工作的动机背景是什么，做了什么，有什么技术原理，解决了什么问题，其意义是什么）

动机背景

近些年出现了一种新的硬件技术，称为“存储类内存（Storage Class Memory）”，简称SCM，它的出现使得人们重新思考原先被隔离开的内存-硬盘的存储结构，SCM结合了传统存储介质经济、容量大、非易失的特点和 DRAM 低延迟、可字节访问的特点，在未来可能SCM会取代DRAM作为一个统一的内存，即同时作为主存和存储器，但是现有的数据结构还不能很好地支持它满足这些需求。

内容（做了什么）

本文提出了一种新的混合SCM-DRAM持久并发B树，即FPTree，它与基于DRAM的并行B树具有相似的性能,介绍了FPTree的详细结构和具体的操作，可以保证从任意点的故障恢复到一个一致性的状态而不丢失信息，除此之外FPTree的访问速度能够与DRAM相当，对不同的SCM设备的延迟有很好的弹性，同时也能支持高并发的场景。目的是改进传统的应用在主存的B-Tree索引结构，因为作者认为传统的索引技术没能满足新硬件特性下的一致性要求，例如对比于DRAM，SCM的延迟更高，写速度比读速度慢很多，除此之外，设计持久化的数据结构、保证数据一致性很难，因为SCM是通过易失的CPU cache进行访问的，软件对于此的控制力很小。简而言之，如果需要更好的性能，就需要设计更适合的数据结构，于是作者设计了FPTree索引，希望可以达到能够相当于DRAM的性能，同时解决一些关于持久化内存泄漏和数据恢复的问题。

技术原理

- Fingerprinting
- Selective Persistence
- Selective Concurrency
- Sound programming model

解决的问题

将索引数做成Persistent，concurrent B+ Tree，可以将数据持久化而且与那些易失性树结构相比，没有增加明显的开销。改进了传统的应用在主存的B-Tree索引结构。传统的主存B-树实现不能满足这种用例所需的一致性要求。解决了在第二节中发现的SCM编程挑战。

意义

FPTree的性能比具有不同SCM延迟的最先进的持久树高出8.2倍。此外，FPTree在具有88逻辑的机器上具有很好的扩展性在使用完全瞬态数据stru时，其性能开销几乎可以忽略不计。而且FPTree使用硬件事务内存(HTM)来处理TH内部节点的并发性和细粒度锁来处理叶节点的并发性.选择性并发巧妙地解决了单片机所要求的HTM和持久化原语的明显不兼容性。同时，它充分利用了SCM的能力，同时表现出与传统的瞬态B-树相似的性能。

SCM硬件有什么特性？与普通磁盘有什么区别？普通数据库以页的粒度读写磁盘的方式适合操作SCM吗？

硬件特性

SCM是通过一个长的波动链(如图1所示)访问的，其中包括存储缓冲区、CPU缓存和内存控制器缓冲区，所有这些软件几乎没有控制。SNIA[3]建议使用SCMEAW文件系统管理SCM，该文件系统允许应用层使用mmap直接访问SCM，从而启用加载/存储语义。因此，SCM WRI的订购和耐久性没有软件的努力就不能保证TES。当程序重新启动时，它会使用一个新的地址空间来执行此操作，该地址空间会使所有存储的虚拟指针无效。

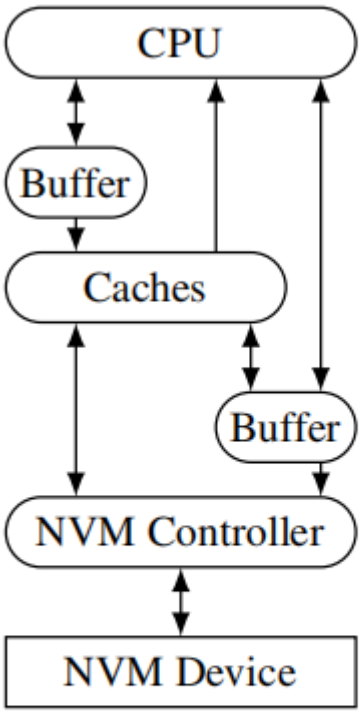


Figure 1: Volatility chain in x86-like processors.

与普通磁盘的区别

SCM的持久性没有软件的支持就不能保证TES，在SCM中内存分配是持久的，内存泄漏也是持久的，因此，与DRAM相比，内存泄漏对SCM的影响更大。

普通数据库以页的粒度读写磁盘的方式适合操作SCM吗?

不适合。由于对SCM的写入具有字粒度，因此可能会发生部分写入。事实上，如果在写入大于支持的p原子写入大小的数据时发生了错误，就不能保证ho。W编写了大量数据。与并发性意义上的原子相反，我们指的是在一个CPU周期内执行的p原子写入，即不受部分写入问题影响的写入。向广告针对这个问题，我们使用可以用p原子写的标志来指示是否完成了更大的写操作。在这项工作中，我们假设只有8字节的写入是p原子的。

操作SCM为什么要调用CLFLUSH等指令?

(写入后不调用，发生系统崩溃有什么后果)

SCM WRI的订购和耐久性没有软件的支持就不能保证TES。为了解决这个问题，使用了类似于最新技术的持久化原语，即CLFLUSH、MFENCE、SFENCE和非时态写入。CLFLUSH将缓存行从缓存中移除，并将其内容写回内存。当发出MFENCE时，所有挂起的加载和存储内存操作都在程序继续进行之前完成。一个s对于非时态写入，它们绕过缓存，并被缓冲在一个特殊的缓冲区中，当缓存满或发出MFENCE时。此外，硬件供应商还宣布了新的指令，以提高SCM编程性能。例如，Intel宣布了CLFLUSHOPT(优化版本)。(CLFLUSH)、PCOMMIT和缓存行写(Clwb)指令[2]。当发出PCOMMIT时，所有从缓存中被逐出的挂起的写操作都是持久的。与CLFLUSH相反，clwb有不是删除缓存行，而是简单地将其写回，这会在缓存行被写回后不久被重新使用时带来显着的性能提高。在这部作品中，我们假设存在函

数的CE持久化，实现了使数据持久的最有效方法。在我们的评估系统中，这个函数可以对应于两个MFENCEs包装的CLFLUSH，因为只有MFence可以命令CLFLUSH[2]，也可以是非时态写入，后面是MFENCE。写入后不调用，系统奔溃有可能造成数据的丢失。

FPTree的指纹技术有什么重要作用?

从理论上讲，指纹技术比wBTree和NV-Tree具有更好的性能。

未分类的叶子需要在SCM中进行昂贵的线性扫描。为了获得更好的性能，使用了指纹技术。指纹是叶键的一个字节散列，连续存储。通过在搜索过程中首先扫描指纹，指纹充当过滤器，以避免探测具有与搜索钥匙。我们只考虑唯一键的情况，这是经常发生的情况。n实践中可接受的假设。作者们证明，在一个成功的搜索过程中，使用指纹技术，期望的叶内键探针数量等于一个。

为了保证指纹技术的数学证明成立，哈希函数应如何选取?

（哈希函数生成的哈希值具有什么特征，能简单对键值取模生成吗？）

哈希函数生成了均匀分布的指纹。不能简单的取模生成。

设m为叶中的条目数，n为可能的散列值数(n=256表示on)。

计算指纹在叶子中出现的预期次数，表示为E[K]，这相当于指纹数组中的哈希碰撞次数加上1(假设搜索键存在)：e[K]=xmi=1i·P[K=i]，其中P[K=i]是搜索指纹至少出现一次的概率。

持久化指针的作用是什么？与课上学到的什么类似?

作用：允许重新启动SCM，持久性指针在失败期间仍然有效，它们用于在重新启动时刷新易失性的。

与我们上课学到的非易失性内存（硬件特性上），可持久化线段树（数据结构）有点相似：

- ROM（Read-only memory，只读内存）
    - PROM（Programmable read-only memory，可编程只读内存）
    - EAROM（Electrically alterable read only memory，电可改写只读内存）
    - EPROM（Erasable programmable read only memory，可擦可编程只读内存）
    - EEPROM（Electrically erasable programmable read only memory，电可擦可编程只读内存）
- Flash memory（闪存）