课程大作业 4

1. 实验目的

通过测试发现虚拟化的性能损耗（与原生性能对比），通过阅读文献等途径，修改开源软件的相关代码，降低虚拟化的性能损耗。

2. 实验内容

2.1 性能损耗

首先我们先进行性能测试并与原生性能进行对比，找到性能损耗的相关部分。回顾到，在作业二中，我们进行了了二次虚拟的性能测试，并且本次实验中我们测试了一次虚拟的实验性能，我们将实验结果在附录进行展示，下面提取出关键的性能对比：

	二次虚拟性能	一次虚拟性能
CPU	309.78	459.08
Memory	8223.43	14475.60
Thread	17502	52844
File IO	288.48 / 192.32 / 614.15	6694.88 / 4463.09 / 14321.47

注：CPU的统计标准：Events per second，Memory的统计标准：MiB/sec，Thread的统计标准：同等时间内的total number of events，FIle IO的统计标准：File operations, reads/writes/fsyncs，单位为s。

题目要求我们测试发现虚拟化的性能损耗（与原生性能相比），在这里，我们将一次虚拟作为原生性能，在一次虚拟的主机上通过 qemu 创建二次虚拟，并将其作为题目中提到的虚拟化性能损耗。通过上述的实验结果可以发现，二次虚拟后的实验性能比一次虚拟后的性能各个部分都有较大的损耗，因此我们认为通过修改此时的 qemu 代码得到的性能提升同样能够迁移到主机和一次虚拟（通过 qemu）上的性能损耗。

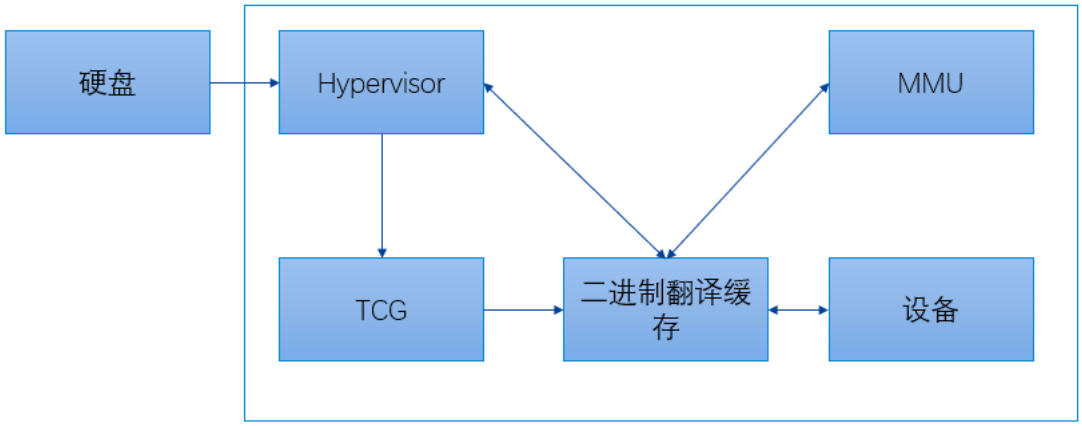
因此下面我们就通过修改 qemu 代码来达到二次虚拟上的性能提升。

观察到，qemu 作为开源软件，虽然能够调用 kvm 以及相关的开源软件比如 xen，能够较好的实现出虚拟化的效果，但是其虚拟后的性能与主机相比而言仍然有较大的损耗，这也是近几年来 qemu 依然作为开源社区活跃在一部分的群体中，希望通过修改源代码添加新的性能或者改进算法来降低虚拟化的性能损耗。

基于 qemu 关于以上测试的各个方面（cpu、memory、thread、File IO）都有较大的损耗，因此理论上提升了任意其中一个方面都能算作降低了虚拟化的性能。下面我将一步步介绍如何通过修改源代码来降低关于文件系统方面的性能损耗。

2.2 qemu 整体框架

qemu 的宏观架构如下图所示，由几个基本的组件组成



如图所示，QEMU 由以下几个部分组成：

- Hypervisor 控制仿真
- Tiny Code Generator（TCG）在虚拟机器代码和宿主机代码之间进行转换。
- 软件内存管理单元（MMU）处理内存访问。
- 磁盘子系统处理不同的磁盘映像格式
- 设备子系统处理网卡和其他硬件设备

由于本次实验主要是针对 memory 部分进行优化，因此下面我们先介绍下内存是如何进行虚拟的，对该部分有了整体上的把握后，我们再对源代码直接进行解读。

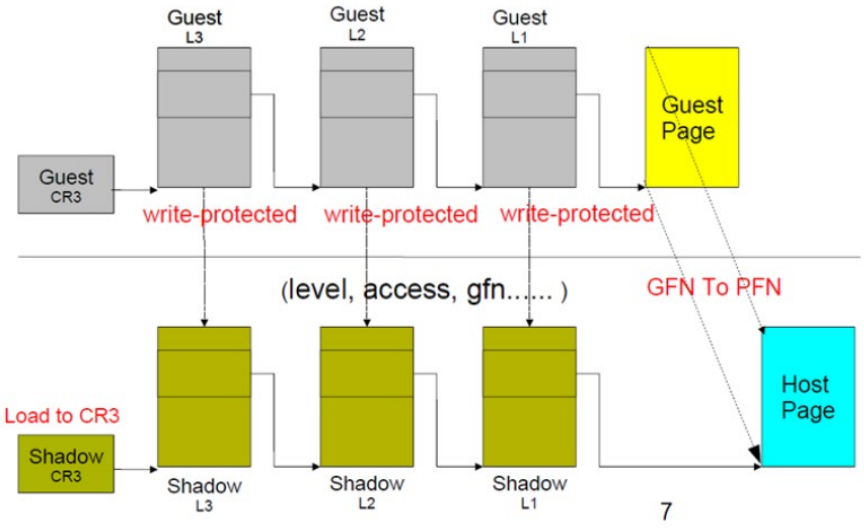
在虚拟机中一共有四个地址：

- GVA（Guest Virtual Address） 虚拟机程序所访问的地址
- GPA（Guest Physical Address）Guest OS 所“认为”的物理地址
- HVA（Host Virtual Address）Host OS 上的程序所访问的地址（包括 Hypervisor）
- HPA（Host Physical Address）真正的物理地址，用于索引 DRAM 上的数据

对应的也有两种访问方式：

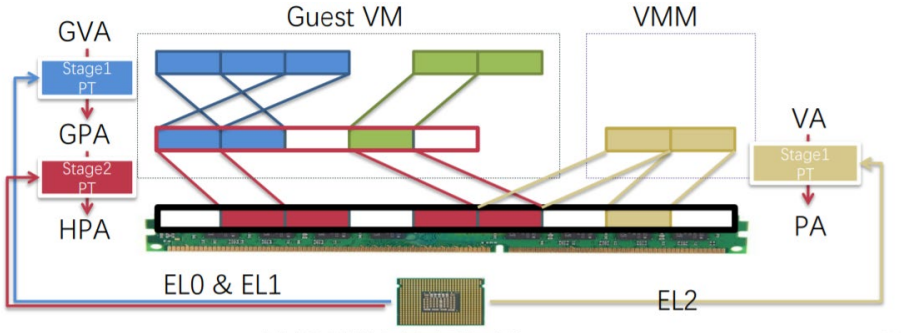
硬件直接访问 GVA -> HPA,该方法为 SPT，或者先访问 Guest OS 的页表，再访问 GPA -> HPA，该方法为 EPT，不过要注意的是，硬件（MMU，TLB）都会先访问 Host OS 页表来找到 HVA 所对应的 HPA。

在 SPT 中，Guest OS“认为”它建立了从 GVA->GPA 的映射，并且硬件会根据改页表寻址，但实际上 Hypervisor 截取了 Guest OS 对页表的修改，并将真实的页表改为 GVA->HPA 的映射，如下图所示，下面黄颜色的页表即为影子页表（Shadow Page Table）。



该方法可以从 GVA->HPA 一步到位，具有较大的灵活性，但是同样缺点也很明显，每一个 Guest 进程都需要有一个 SPT，这样会造成大量的内存消耗，并且每一次页表修改都会导致 VMExit 和 TLB flush，降低了性能。

而在 EPT 中，则存在两个阶段的翻译，第一阶段是将 GVA->GPA，硬件根据 Guest 页表进行翻译，完全不受 Hypervisor 控制；第二阶段是将 GPA->HPA，硬件根据 EPT 进行翻译，受 Hypervisor 控制，如下图所示：



而当 VT-x 硬件发现 GPA 没有对应的项，或者 CPU 违反了 EPT 表项所指定的规则（如写只读页），它会向 CPU 注入一种特殊的 VMExit：EPT Violation。因此我们可以认为，当发生内存调度中出现的问题时，虚拟化都会模拟 Host 主机内存中的解决办法来解决虚拟化中的问题。

2.3 qcow2 文件

再对源码分析前，我们需要先对 qcow2 文件有一定的了解，之后才能对源代码中相关部分的内容有更清晰的认识。

qcow2 镜像格式是 QEMU 模拟器支持的一种磁盘镜像。它也是可以用一个文件的形式来表示一块固定大小的块设备磁盘。与普通的 raw 格式的镜像相比，有以下特性：

- 更小的空间占用，即使文件系统不支持空洞(holes)；
- 支持写时拷贝（COW, copy-on-write），镜像文件只反映底层磁盘的变化；
- 支持快照（snapshot），镜像文件能够包含多个快照的历史；
- 可选择基于 zlib 的压缩方式
- 可以选择 AES 加密

qcow2 镜像文件是由多个固定大小的单元组织构成，这些单元被称为(host clusters)。无论是实际用户数据（guest data）还是镜像的元数据（metadata），都在一个 cluster 单元中进行存储。同样的，用户所见到的虚拟磁盘也是被分割为多个同样大小的 clusters。qcow2 里所有的数都是 Big Endian 的。

对于每一个 host cluster，qcow2 维护了一个 refcount 表，当 refcount 为 0 时，表示该 cluster 是未分配的，1 表示是在使用的，>=2 时表示在被使用，并且所有的写操作都要进行 COW(copy on write)操作。注意到，qcow2 采用了两层表来维护管理 refcounts，第一层叫 refcount table，是可变大小的（refcount table 的 size 存储在 header 里），refcount table 的每一项覆盖多个 cluster，当然，在镜像文件中 refcount table 是连续存储的。refcount table 包含了多个指针，指向了第二层结构体，第二层结构被称为 refcount block，一个 refcount block 在大小上就是一个 cluster。根据镜像偏移量 offset，我们可以获得某个 cluster 对应引用计数的方法：

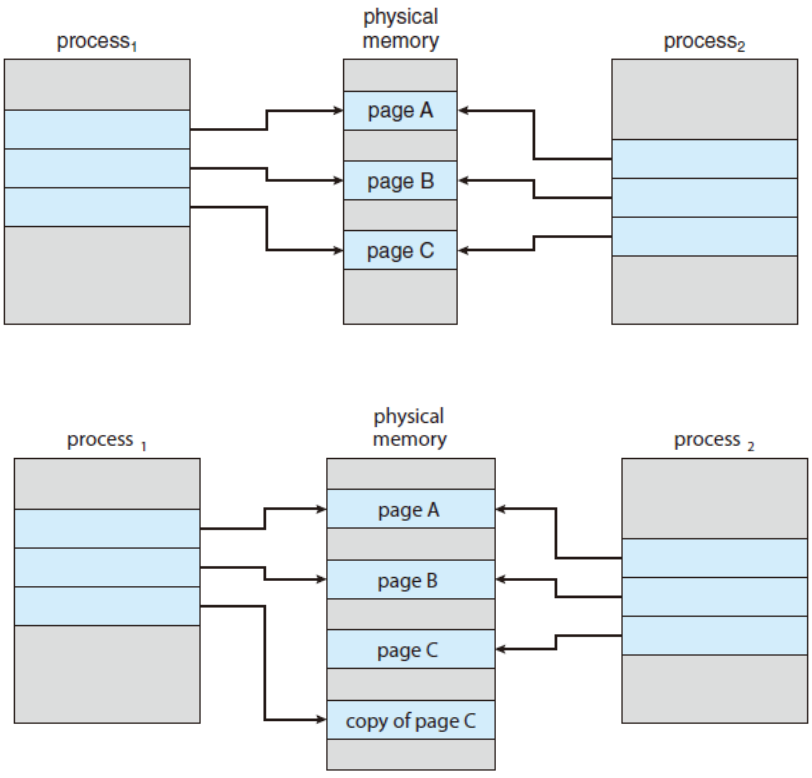
```
refcount_block_entries = (cluster_size * 8 / refcount_bits)

refcount_block_index = (offset / cluster_size) % refcount_block_entries
refcount_table_index = (offset / cluster_size) / refcount_block_entries

refcount_block = load_cluster(refcount_table[refcount_table_index]);
return refcount_block[refcount_block_index];
```

而这就类似于文件系统的段页式管理（便于理解）。注意到，qcow 文件格式中 cow 即代表 Copy-On-Write，即说明 qcow 文件中写复制是其核心算法。

根据第十版的操作系统上关于 COW 的定义，我们可以得知，当父进程通过 fork()函数产生了一个与父进程相同的子进程，如果按照传统的做法，会直接将父进程的数据拷贝到子进程中，拷贝完后，父进程和子进程的数据段和堆栈是相互独立的。但是考虑到往往子进程

在创建完之后会马上调用 exec()函数来实现自己想要实现的功能，那么这时候复制过去的数据都是没用的，(因为 exec()函数会清空原来的数据），因此就有了 COW 技术。

原理也很简单，fork 创建出的子进程，与父进程共享内存空间。也就是说，如果子进程不对内存空间进行写入操作的话，内存空间中的数据并不会复制给子进程，这样创建子进程的速度就很快了，并且如果尝试写入数据，系统就会复制创建一个新的数据段，所有的操作将在上面进行。如下图所示：



以上就是需要事先了解的一些知识，下面我们将直接对源代码进行解读并且根据上述我们了解的知识来修改源代码提升性能。

2.4 源码修改

在源码/block/qcow2 中可以发现对于 qcow2 文件的写入操作，由于对于 qemu 创建的虚拟机来说，我们都是先用 qemu-img 创建一个文件磁盘，然后在该磁盘上装入操作系统（比如 ubuntu），因此我们可以认为，qcow2 文件对于虚拟机而言，正如磁盘对于主机而言。因此对 qcow2 文件的读写即可认为是对磁盘的读写，因此对磁盘读写的那一套逻辑也可以在这里使用。

注意到源码当中，当有新的写入请求导致需要重新分配新的空间时，QEMU 会首先尝试查看该 cluster 中写入区域之外的部分是不是仅包含 0，即为初始状态。如果是在上述的情

况下，代码就无需执行正常的写时复制操作并显式地将 0 buffer 写入到磁盘中，而是使用带 BDRV_REQ_NO_FALLBACK 的 pwrite_zeroes()将整个 cluster 有效地归零。

虽然这可以极大地提升性能，但是注意到这只发生在这块将要被写入的区域在之前是从未被分配过的。此时，Zero cluster(QCOW2_CLUSTER_ZERO_*)被视作普通的 cluster，因此分配的速度较慢。其代码如下所示：

判断该块 cluster 是否被分配函数：

```
static bool is_unallocated(BlockDriverState *bs, int64_t offset, int64_t bytes)
{
    int64_t nr;
    return !bytes ||
        (!bdrv_is_allocated_above(bs, NULL, false, offset, bytes, &nr) &&
         nr == bytes);
}
```

判断该块 cluster 是否为 0 cluster：

```
static int is_zero_cow(BlockDriverState *bs, QCowL2Meta *m)
{
    /*
     * This check is designed for optimization shortcut so it must be
     * efficient.
     * Instead of is_zero(), use is_unallocated as it is faster (but not
     * as accurate and can result in false negatives).
     */

    return is_unallocated(bs, m->offset + m->cow_start.offset,
                          m->cow_start.nb_bytes) &&
           is_unallocated(bs, m->offset + m->cow_end.offset,
                          m->cow_end.nb_bytes);
}
```

注意到，在判断是否为 0 cluster 时，函数返回的是两个区域之前是否被分配过的交，虽然理论上来说未被分配过则一定为 0 cluster，但是也可能存在分配过但是后续又被清 0 的区域，该部分代码则未进行考虑。虽然正如注释所说，使用 is_unallocated 函数会比较快，但是结果可能不完全正确并且会导致 false negatives，而且注意到，该函数是作为下面分配空间时判断该 cluster 是否为 0 cluster 的一个判断函数，该判断函数影响着后续的操作，即使用带 BDRV_REQ_NO_FALLBACK 的 pwrite_zeroes()将整个 cluster 有效地归零。

在这里，我们很容易就能够想到，虽然判断函数确实快了，但是该判断函数影响的是一个优化的操作是否被执行，该优化的操作可以极大的提升性能，综合考虑 tradeoff，我们可以轻易地判断，在判断函数上多花一点微不足道的时间，可以换来后续更多的优化操作，这样总体而言性能还是得到了提升。

因此，我们有理由，也有依据地认为这部分的代码可以修改来得到更好的性能。

下面为 QEMU 中处理分配空间的函数（部分）：

```
static int handle_alloc_space(BlockDriverState *bs, QCowL2Meta *l2meta)
{
    BDRVQcow2State *s = bs->opaque;
    QCowL2Meta *m;


        if (!is_zero_cow(bs, m)) {
            continue;
        }


        /*
         * instead of writing zero COW buffers,
         * efficiently zero out the whole clusters
         */

        ret = qcow2_pre_write_overlap_check(bs, 0, start_offset, nb_bytes,
                                            true);
```

可以看到，当判断为 is_zero_cow 之后，直接高效地将整个 cluster 归零。

而 is_unallocated 之所以会发生上述的情况,是因为它使用的是底层函数 bdrv_is_allocated_above()，而该函数是定义在/block/io.c 文件中。但是在该文件中同样存在着 bdrv_common_block_status_above()函数,该函数同样也可以用来判断该 cluster 是否为 0 cluster，并且没有该区域必须为之前完全没被分配的前提。虽然速度上不如前一个函数，但是正如上述所述，我们完全可以使用相对较慢的函数来达到整体性能上的提升。因此我们利用该函数重新包装了一个判断是否为 0 cluster 函数，并将其运用于判断函数 is_zero_cow()函数中。

函数 bdrv_common_block_status_above（）：

```
static int bdrv_common_block_status_above(BlockDriverState *bs,
                                          BlockDriverState *base,
                                          bool want_zero, int64_t offset,
                                          int64_t bytes, int64_t *pnum,
                                          int64_t *map,
                                          BlockDriverState **file)
{
    BdrvCoBlockStatusData data = {
        .bs = bs,
        .base = base,
        .want_zero = want_zero,
        .offset = offset,
        .bytes = bytes,
        .pnum = pnum,
        .map = map,
        .file = file,
    };

    return bdrv_run_co(bs, bdrv_block_status_above_co_entry, &data);
}
```

函数 bdrv_is_allocated_above()：

```
/*
 * Given an image chain: ... -> [BASE] -> [INTER1] -> [INTER2] -> [TOP]
 *
 * Return 1 if (a prefix of) the given range is allocated in any image
 * between BASE and TOP (BASE is only included if include_base is set).
 * BASE can be NULL to check if the given offset is allocated in any
 * image of the chain.  Return 0 otherwise, or negative errno on
 * failure.
 *
 * 'pnum' is set to the number of bytes (including and immediately
 * following the specified offset) that are known to be in the same
 * allocated/unallocated state.  Note that a subsequent call starting
 * at 'offset + *pnum' may return the same allocation status (in other
 * words, the result is not necessarily the maximum possible range);
 * but 'pnum' will only be 0 when end of file is reached.
 *
 */
int bdrv_is_allocated_above(BlockDriverState *top,
                            BlockDriverState *base,
                            bool include_base, int64_t offset,
                            int64_t bytes, int64_t *pnum)
{
```

我们基于 bdrv_common_block_status_above（）函数给出新的判断 0 cluster 的函数，如下图所示：

```
int coroutine_fn bdrv_co_is_zero_fast(BlockDriverState *bs, int64_t offset,
                                      int64_t bytes)
{
    int ret;
    int64_t pnum = bytes;

    if (!bytes) {
        return 1;
    }

    ret = bdrv_common_block_status_above(bs, NULL, false, offset,
                                         bytes, &pnum, NULL, NULL);

    if (ret < 0) {
        return ret;
    }

    return (pnum == bytes) && (ret & BDRV_BLOCK_ZERO);
}
```

我们首先检查@bs（以及它的 backing chain）去看看通过@offset 和@bytes 定义的区域是否是 0 cluster，如果是的，则返回 1，其他情况返回 0，出错时则返回负的 error number。同样也需要注意到，该函数也是相较于正确率更强调速度，因此有可能返回 0 时并不保证一定为非 0 cluster。

然后在 is_zero_cow()函数中调用我们新添的函数。不过注意到返回的值与原来相比有所不同，返回 1 代表为 0 cluster，返回 0 代表不是，返回小于 0 代表出错。

```
/*
    Return 1 if the COW regions read as zeroes, 0 if not, <0 on error.
    Note that returning 0 does not guarantee non-zero data.
*/
static int is_zero_cow(BlockDriverState *bs, QCowL2Meta *m)
{
    /*
     * This check is designed for optimization shortcut so it must be
     * efficient.
     * Instead of is_zero(), use bdrv_co_is_zero_fast() as it is faster (but not
     * as accurate and can result in false negatives).
     */

    int ret = bdrv_co_is_zero_fast(bs,m->offset+m->cow_start.offset,m->cow_start.nb_bytes);

    if (ret <= 0) {
       return ret;
    }
    return bdrv_co_is_zero_fast(bs,m->offset+m->cow_end.offset,m->cow_end.nb_bytes);

    // return is_unallocated(bs, m->offset + m->cow_start.offset,
    //                       m->cow_start.nb_bytes) &&
    //        is_unallocated(bs, m->offset + m->cow_end.offset,
    //                       m->cow_end.nb_bytes);
}
```

然后再在调用该判断函数的 handle_alloc_space 中修改一下判断逻辑即可。

```
// if (!is_zero_cow(bs, m)) {
//     continue;
// }
ret = is_zero_cow(bs,m);
if (ret < 0 ) {
   return ret;
} else if (ret == 0) {
   continue;
}
```

以上便是本次修改源代码的全部过程，而具体的修改详情我会以 github 中的版本 diff 来形象地展示。除了上述的修改外，还需要在 header 文件中将我们新添的函数名加上去，附录中也将其展示出来。

3. 实验结论

下图为修改源代码后重新编译跑出来的结果，从结果上看的话，相较于原来的二次虚拟，file IO 操作的 reads/s, writes/s, fsyncs/s 从原来的 288.48/192.32/614.15 变为现在的 339.15/225.90/716.49，我们可以看到，file IO 上各个方面都相较于原来的数值有了较大的提升，不过相比于主机性能依然有较大的差异。

不过该变化也是可解释的，注意到，我们仅仅只是改变了当为 0 cluster 时采用带 BDRV_REQ_NO_FALLBACK 的 pwrite_zeroes()将整个 cluster 有效地归零方法的频率，但是在测试过程中往 0 cluster 写入数据的操作就不多，大部分还是基于现存的 cluster 来进行读写。

如果能够在仅含 0 cluster 上进行读写操作，那么理论上能够达到原来速率 3-4 倍。

```
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Extra file open flags: 0
128 files, 24MiB each
3GiB total file size
Block size 16KiB
Number of IO requests: 0
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Initializing worker threads...

Threads started!


File operations:
    reads/s:                      339.15
    writes/s:                     225.90
    fsyncs/s:                     716.49

Throughput:
    read, MiB/s:                  5.30
    written, MiB/s:               3.53

General statistics:
    total time:                          10.0030s
    total number of events:              12821

Latency (ms):
         min:                                    0.00
         avg:                                    3.10
         max:                                  157.59
         95th percentile:                       11.87
         sum:                                39747.53

Threads fairness:
    events (avg/stddev):           3205.2500/44.45
    execution time (avg/stddev):   9.9369/0.03
```

而之后我们同样测试了 memory 的性能，因为考虑到 memory 的性能测试即也是对内存的读写操作，并且最开始内存为空时数据也要从磁盘中读取得来，因此我们有理由认为 memory 的性能应该也有所提升。而从实验结果上看（下图），memory 性能也有了不小的提升，从原来的 8223.43 MiB/sec 变为现在的 10890.66 MiB/sec。

```
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 8KiB
  total size: 102400MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 13107200 (1394003.84 per second)

102400.00 MiB transferred (10890.66 MiB/sec)


General statistics:
    total time:                          9.4015s
    total number of events:              13107200

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                   52.01
         95th percentile:                        0.00
         sum:                                29974.32

Threads fairness:
    events (avg/stddev):           3276800.0000/0.00
    execution time (avg/stddev):   7.4936/0.14
```

4. 实验难点和感悟

本次实验其实在作业二时就有所覆盖，不过此时修改源码仅仅是也只是为了修改而修改（比如加个 print），但是一旦要开始降低性能损耗而修改时，整个作业的难度也就不同了。

为了了解性能损耗，我们首先得知道哪个部分的性能不足，确定到具体的目标后（比如 file IO），需要首先了解原本操作系统中关于该部分的知识（比如段页式管理，TLB），知道这部分有哪些比较优秀的算法，之后需要去了解 QEMU 代码的整体框架，因为某一个性能往往是许多软硬件协同得出的效果，只有从整体上把握才能对后续的修改有个清晰的逻辑，最后我们还需要了解 QEMU 源码的架构，QEMU 源码部分逻辑比较清晰，但是很难把握整体的思路。

好在 QEMU 源码中对每个函数都有比较清晰的注释，我们不需要去推测该函数的功能，而只需“对症下药”即可，这大大缓解了修改源码的压力。

但是对于每一个 c 文件里面至少有 3000-4000 行，想要在里面搜寻到特定的功能实在是过于困难，网上关于 QEMU 源码的分析又十分少，对于新手来说可谓是十分的不友好了。好在本次作业的时间周期足够的长，我也能有足够长的时间去浏览一遍里面的函数。

但是其实浏览的途中往往只会被动的去接受，很难再尝试读懂的同时就能想到这部分有哪些缺陷。上面的修改源码的思路很大程度是源于注释中整个函数前有一个的 “为了追求速度”这一个比较醒目的牺牲，但是里面又有个注释“这样能够极大地提高性能”并且这部分的内容和前面又相配的不是很好，我猜测应该是后续有人基于此部分代码 commit 的部分，但是又没有考虑完整，因此才会显得比较矛盾，但是这样也才给了我修改的灵感。

通过本次的作业，我也理解了开源社区中每一位贡献者的不易，也倾佩那些依然默默地为开源软件一直增添新功能的贡献者们，只有自己实际尝试过才懂得他们的不易。

以上就是本次作业的全部内容和感想，谢谢。

5. 附录

2.1 性能损耗的实验结果

2.1.1 二次虚拟性能

- cpu performance benchmark test:

```
sysbench --test=cpu --num-threads=4 --cpu-max-prime=20000 run
```

- Thread benchmark test:

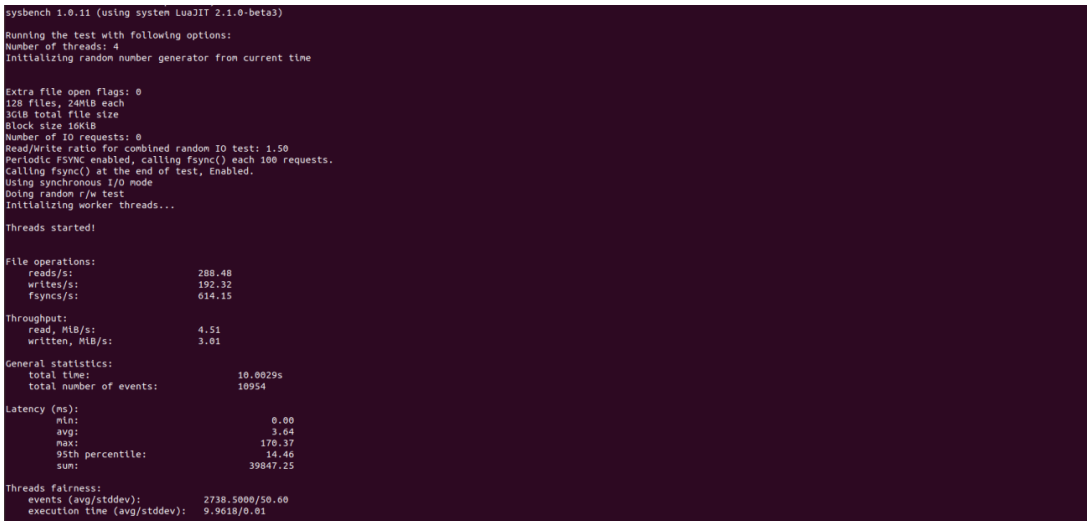
```
sysbench --test=threads --num-threads=4 --thread-yields=100 --thread-locks=2 run
```

- Memory performance benchmark test:

```
sysbench --test=memory --num-threads=4 --memory-block-size=8K --memory-total-size=100G
run
```

- File IO benchmark test:

```
sysbench --test=fileio --num-threads=4 --file-total-size=3G --file-test-mode=rndrw
prepare
sysbench --test=fileio --num-threads=4 --file-total-size=3G --file-test-mode=rndrw run
sysbench --test=fileio --num-threads=4 --file-total-size=3G --file-test-mode=rndrw
cleanup
```



2.1.2 原生性能（一次虚拟）

- CPU speed

```
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Prime numbers limit: 20000

Initializing worker threads...

Threads started!

CPU speed:
    events per second:   459.08

General statistics:
    total time:                          10.0057s
    total number of events:              4594

Latency (ms):
         min:                                    1.95
         avg:                                    8.70
         max:                                   47.12
         95th percentile:                       17.95
         sum:                                39970.70

Threads fairness:
    events (avg/stddev):           1148.5000/0.50
    execution time (avg/stddev):   9.9927/0.00
```

- Memory

```
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Running memory speed test with the following options:
  block size: 8KiB
  total size: 102400MiB
  operation: write
  scope: global

Initializing worker threads...

Threads started!

Total operations: 13107200 (1852876.20 per second)

102400.00 MiB transferred (14475.60 MiB/sec)


General statistics:
    total time:                          7.0728s
    total number of events:              13107200

Latency (ms):
         min:                                    0.00
         avg:                                    0.00
         max:                                   32.26
         95th percentile:                        0.00
         sum:                                23828.60

Threads fairness:
    events (avg/stddev):           3276800.0000/0.00
    execution time (avg/stddev):   5.9572/0.05
```

- Thread

```
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time


Initializing worker threads...

Threads started!


General statistics:
    total time:                          10.0003s
    total number of events:              52844

Latency (ms):
         min:                                    0.04
         avg:                                    0.76
         max:                                   16.86
         95th percentile:                        1.55
         sum:                                39975.90

Threads fairness:
    events (avg/stddev):           13211.0000/163.37
    execution time (avg/stddev):   9.9940/0.00
```

- File IO

```
Extra file open flags: (none)
128 files, 24MiB each
3GiB total file size
Block size 16KiB
Number of IO requests: 0
Read/Write ratio for combined random IO test: 1.50
Periodic FSYNC enabled, calling fsync() each 100 requests.
Calling fsync() at the end of test, Enabled.
Using synchronous I/O mode
Doing random r/w test
Initializing worker threads...

Threads started!


File operations:
    reads/s:                      6694.88
    writes/s:                     4463.09
    fsyncs/s:                     14321.47

Throughput:
    read, MiB/s:                  104.61
    written, MiB/s:               69.74

General statistics:
    total time:                          10.0087s
    total number of events:              254557

Latency (ms):
         min:                                    0.00
         avg:                                    0.16
         max:                                   11.64
         95th percentile:                        0.55
         sum:                                39736.38

Threads fairness:
    events (avg/stddev):           63639.2500/302.06
    execution time (avg/stddev):   9.9341/0.00
```

2.4 源码修改

```
26 ■■■■■ io.c
@@ -2544,6 +2544,32 @@ int bdrv_block_status(BlockDriverState *bs, int64_t offset, int64_t bytes,
                                offset, bytes, pnum, map, file);
 }

+/*
+ * Check @bs (and its backing chain) to see if the range defined
+ * by @offset and @bytes is known to read as zeroes.
+ * Return 1 if that is the case, 0 otherwise and -errno on error.
+ * This test is meant to be fast rather than accurate so returning 0
+ * does not guarantee non-zero data.
+ */
+int coroutine_fn bdrv_co_is_zero_fast(BlockDriverState *bs, int64_t offset,
+                                      int64_t bytes)
+{
+    int ret;
+    int64_t pnum = bytes;
+
+    if (!bytes) {
+        return 1;
+    }
+
+    ret = bdrv_common_block_status_above(bs, NULL, false, offset,
+                                         bytes, &pnum, NULL, NULL);
+
+    if (ret < 0) {
+        return ret;
+    }
+
+    return (pnum == bytes) && (ret & BDRV_BLOCK_ZERO);
+}
+
 int coroutine_fn bdrv_is_allocated(BlockDriverState *bs, int64_t offset,
                                    int64_t bytes, int64_t *pnum)
```

28 qcow2.c

```
@@ -2399,20 +2399,26 @@ static bool is_unallocated(BlockDriverState *bs, int64_t offset, int64_t bytes)
     Return 1 if the COW regions read as zeroes, 0 if not, <0 on error.
     Note that returning 0 does not guarantee non-zero data.
 */
-
 static int is_zero_cow(BlockDriverState *bs, QCowL2Meta *m)
 {
     /*
      * This check is designed for optimization shortcut so it must be
      * efficient.
-     * Instead of is_zero(), use is_unallocated as it is faster (but not
+     * Instead of is_zero(), use bdrv_co_is_zero_fast() as it is faster (but not
      * as accurate and can result in false negatives).
      */
-    return is_unallocated(bs, m->offset + m->cow_start.offset,
-                          m->cow_start.nb_bytes) &&
-           is_unallocated(bs, m->offset + m->cow_end.offset,
-                          m->cow_end.nb_bytes);
+    int ret = bdrv_co_is_zero_fast(bs,m->offset+m->cow_start.offset,m->cow_start.nb_bytes);
+
+    if (ret <= 0) {
+      return ret;
+    }
+    return bdrv_co_is_zero_fast(bs,m->offset+m->cow_end.offset,m->cow_end.nb_bytes);
+
+    // return is_unallocated(bs, m->offset + m->cow_start.offset,
+    //                       m->cow_start.nb_bytes) &&
+    //        is_unallocated(bs, m->offset + m->cow_end.offset,
+    //                       m->cow_end.nb_bytes);
 }

 static int handle_alloc_space(BlockDriverState *bs, QCowL2Meta *l2meta)
@@ -2438,8 +2444,14 @@ static int handle_alloc_space(BlockDriverState *bs, QCowL2Meta *l2meta)
             continue;
         }

-        if (!is_zero_cow(bs, m)) {
-            continue;
+        // if (!is_zero_cow(bs, m)) {
+        //     continue;
+        // }
+        ret = is_zero_cow(bs,m);
+        if (ret < 0 ) {
+          return ret;
+        } else if (ret == 0) {
+          continue;
         }
```

2 include/block/block.h

```
@@ -508,6 +508,8 @@ int bdrv_is_allocated(BlockDriverState *bs, int64_t offset, int64_t bytes,
 int bdrv_is_allocated_above(BlockDriverState *top, BlockDriverState *base,
                             bool include_base, int64_t offset, int64_t bytes,
                             int64_t *pnum);
+int coroutine_fn bdrv_co_is_zero_fast(BlockDriverState *bs, int64_t offset,
+                                      int64_t bytes);

 bool bdrv_is_read_only(BlockDriverState *bs);
 int bdrv_can_set_read_only(BlockDriverState *bs, bool read_only,
```