基于 shm 恢复机制

1.涉及到的系统 API

1.1 共享内存相关

1.1.1 posix 系列

1.1.1.1 映射概述

posix 共享内存映射方式较为灵活。从映射目标角度分析，操作系统系统提供三种映射方式，分别是映射实体磁盘文件、虚拟内存文件和匿名映射。

1.1.1.2shm_open

打开或创建一个虚拟内存文件，该文件生成位置为 /dev/shm/。此处的虚拟内存文件由 tmpfs(即虚拟内存文件系统)管理，特点为使用多少实际占据多大空间。

注意此虚拟内存文件的打开参数与磁盘文件的打开参数 open 类似。

1.1.1.3shm_unlink

删除一个虚拟内存文件。注意程序在 unmap 之后进程只是 detach 了这份虚拟内存文件，不做删除动作。

1.1.1.4mmap

上文 1.1.1.1 节有提到 posix 的映射目标方式有三种，此三种方式还有四种标志来提供 mmap 不同的映射特性。

标志 MAP_PRIVATE

- 采用了写时复制技术（cow）。
- 在映射内容上的改变，对其他进程不可见，变更不会体现在映射文件上。
- 由于不会回写文件，所以使用量超过物理内存和交换内存总量时会遇到 OOMKILL。

标志 MAP_SHARED

- 在映射内容上的改变，对其他进程可见，变更体现在映射文件上。
- 使用量超过物理内存和交换内存总量时也不会出问题。

标志 MAP_ANONYMOUS

- 不映射实际文件。
- 其内容会自动初始化为 0。
- 由于不会回写文件，所以使用量超过物理内存和交换内存总量时会遇到 OOMKILL。

标志 MAP_NORESERVE

- 强制绕开 mmap 调用时对映射大小的检验。

1.1.1.5 对各标志的测试情况

对物理内存 16G 的系统，mmap16G 内存的各标志测试结果见下图 1：

场景	序列	映射类型	结果
A	1	MAP_PRIVATE	mmap报错
A	2	MAP_PRIVATE + MAP_NORESERVE	mmap成功，在持续写入情况下，遇到OOM Killer
A	3	MAP_SHARED	mmap成功，在持续写入正常
B	4	MAP_PRIVATE	mmap成功，在持续写入情况下，有一个进程会遇到OOM Killer
B	5	MAP_PRIVATE + MAP_NORESERVE	mmap成功，在持续写入情况下，有一个进程会遇到OOM Killer
B	6	MAP_SHARED	mmap成功，在持续写入正常

图 1

1.1.1.6 对 mmap 映射操作系统层面理解

见图 2。

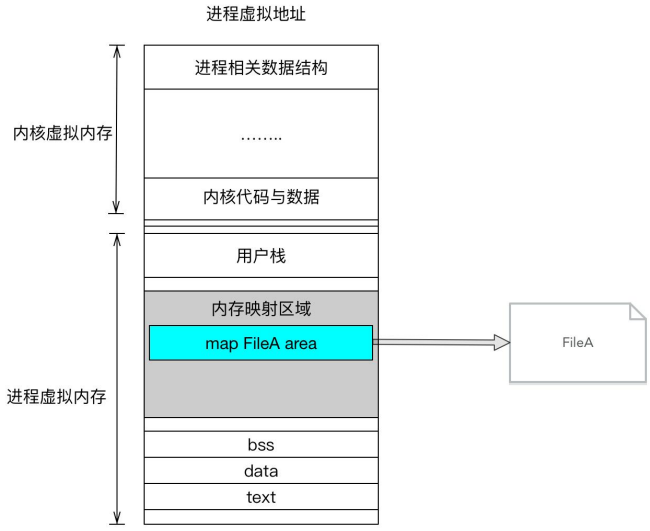

图 2

1.1.1.7 mmap 读写性能与 write/read 比对

- 写部分

  write 方式，过程见图 3。

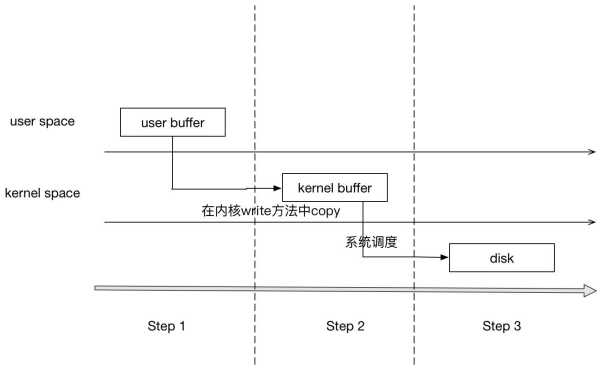

图 3

write 过程简单描述如下：

首先进程(用户态)调用 write 系统调用，并告诉内核需要写入数据的开始地址与长度（告诉内核写入的数据在哪）。接着内核调用 write 方法，将校验用户态的数据，然后复制到 kernel buffer（这里是 Page Cache）。最后由操作系统调用，将脏页回写到磁盘（通常这是异步的）。

mmap 写入过程简单描述如下：

首先进程(用户态)将需要写入的数据直接 copy 到对应的 mmap 地址(内存 copy)，若 mmap 地址未对应物理内存，则产生缺页异常，由内核处理。若已对应，则直接 copy 到对应的物理内存。由操作系统调用，将脏页回写到磁盘（同样通常这是异步的）。

写过程性能预测：

若每次写入的数据大小接近 page size(x86 下 4096)，那么 write 调用与 mmap 的写性能应该比较接近（因为系统调用次数相近）

若每次写入的数据非常小，那么 write 调用的性能应该远慢于 mmap 的性能。

测试结果：

场景：对 2G 的文件进行顺序写入

每次写入大小 | mmap 耗时 | write 耗时
| 1 byte | 22.14s | >300s
| 100 bytes | 2.84s | 22.86s
| 512 bytes | 2.51s | 5.43s
| 1024 bytes | 2.48s | 3.48s
| 2048 bytes | 2.47s | 2.34s
| 4096 bytes | 2.48s | 1.74s
| 8192 bytes | 2.45s | 1.67s
| 10240 bytes | 2.49s | 1.65s

- 读部分

read 方式， 过程见图 4.

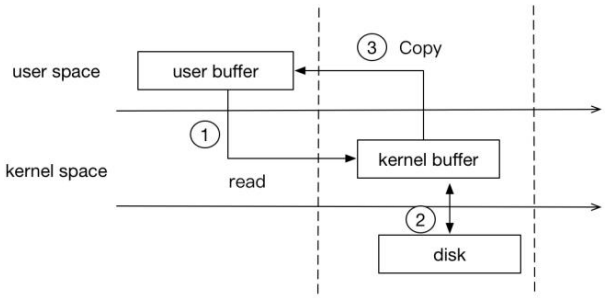

图 4

mmap 方式，过程见图 5.

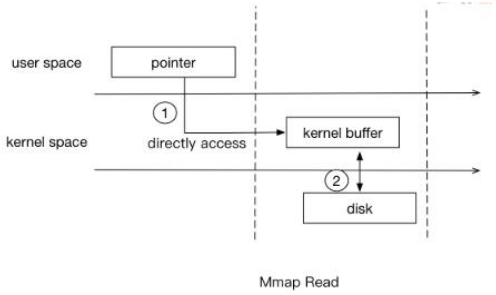

图 5

读性能预测：

从图中可以看出，read 调用确实比 mmap 多一次 copy。因为 read 调用，进程是无法直接访问 kernel space 的，所以在 read 系统调用返回前，内核需要将数据从内核复制到进程指定的 buffer。但 mmap 之后，进程可以直接访问 mmap 的数据(page cache)。

测试结果：

场景：对 2G 的文件进行顺序读取
每次读取大小 | mmap 耗时 | read 耗时
| 1 byte | 8215.4ms | > 300s
| 100 bytes | 86.4ms | 8100.9ms
| 512 bytes | 16.14ms | 1851.45ms
| 1024 bytes | 8.11ms | 992.71ms

| 2048 bytes | 4.09ms | 636.85ms
| 4096 bytes | 2.07ms | 558.10ms
| 8192 bytes | 1.06ms | 444.83ms
| 10240 bytes | 867.88μs | 475.28ms

1.1.2 systemV 系列

1.1.2.1 ftok

此方法根据路径和序号生成一个用于操作 ipc 的唯一 key。

1.1.2.2 shmget

此方式根据 ipc 的唯一 key 获取一块共享内存的 shmid。

标志 IPC_CREAT

返回一个已存在或新创建一个共享内存的 shmid。

标志 IPC_EXCL

配合 IPC_CREAT 使用，如对应共享内存已存在则返回-1，并将 errno 置为 EEXIST。

1.1.2.3 shmat

根据 shmget 返回的 shmid attach 一块共享内存。

1.1.2.4 shmctl

标志 IPC_RMID

此标志相对比较重要，其作用为标记该块共享内存将要被销毁。注意此时不一定被即时销毁，这里使用了引用计数技术，当 attach 这块共享内存的进程数量为 0 的时候才真正执行销毁动作。

1.1.2.5 shmdt

根据 shmid detach 一块共享内存。注意此时该共享内存并没有从操作系统中释放，类比 unmap。

1.1.2.6 控制台外部操作方式

使用 ipcs -m 查看当前激活的 systemV 系列共享内存，例见图 6，

图 6

如上图，此时并没有实际创建 systemV 系列的共享内存。

使用 iprm 命令手动删除对应的 systemV 系列的共享内存。

详细内容请参见 man ipcs。

1.1.3 posix 系列 和 systemV 系列选择

在 posix 系列中如果映射目标为虚拟内存文件或者是磁盘文件的话该映射目标能在用户态可见，而 systemV 系列生成的共享内存在用户态看不到实体文件。

posix 系列的在程序访问非法内存地址时产生的 core 文件，不可查看正常的内存值，会提示 can not access memory。为了避免此情况选用 systemV 系列 API 作为共享内存的首选操

作方式。

注意！虽然如此，但后续查出导致 posix 系列此现象的原因。

即，这里映射的内存是否能被 dump 下来收到两部分控制。第一部分为被映射的这段内存的操作属性，可以由 madvise 系统调用做调整，默认为 MADV_DODUMP(即此段内存可被 dump)，可以用 MADV_DONTDUMP 对映射的内存部分做剪切，具体见 man madvise。第二部分是每个进程自己的 coredump_filter,默认对 mmap 的内存有筛选，具体见 man core 。

1.1.4 映射内存地址问题排查方式

- 使用 cat /proc/进程号/maps 查看映射的地址空间，借此排查内存地址问题。
- gdb 下通过 info proc mappings 命令查看生成当前 core 文件的可执行程序当时映射的共享内存地址空间。例，见图 7：

```
(gdb) info proc mappings
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/cyd/workspace/a.out
            0x600000           0x601000     0x1000        0x0 /home/cyd/workspace/a.out
            0x601000           0x602000     0x1000     0x1000 /home/cyd/workspace/a.out
      0x7f7cdf68b000     0x7f7cdf84d000   0x1c2000        0x0 /usr/lib64/libc-2.17.so
      0x7f7cdf84d000     0x7f7cdfa4d000   0x200000   0x1c2000 /usr/lib64/libc-2.17.so
      0x7f7cdfa4d000     0x7f7cdfa51000     0x4000   0x1c2000 /usr/lib64/libc-2.17.so
      0x7f7cdfa51000     0x7f7cdfa53000     0x2000   0x1c6000 /usr/lib64/libc-2.17.so
      0x7f7cdfa58000     0x7f7cdfa6e000    0x16000        0x0 /usr/local/gcc5.1/lib64/libgcc_s.so.1
      0x7f7cdfa6e000     0x7f7cdfc6d000   0x1ff000    0x16000 /usr/local/gcc5.1/lib64/libgcc_s.so.1
      0x7f7cdfc6d000     0x7f7cdfc6e000     0x1000    0x15000 /usr/local/gcc5.1/lib64/libgcc_s.so.1
      0x7f7cdfc6e000     0x7f7cdfc6f000     0x1000    0x16000 /usr/local/gcc5.1/lib64/libgcc_s.so.1
      0x7f7cdfc6f000     0x7f7cdfd70000   0x101000        0x0 /usr/lib64/libm-2.17.so
      0x7f7cdfd70000     0x7f7cdff6f000   0x1ff000   0x101000 /usr/lib64/libm-2.17.so
      0x7f7cdff6f000     0x7f7cdff70000     0x1000   0x100000 /usr/lib64/libm-2.17.so
      0x7f7cdff70000     0x7f7cdff71000     0x1000   0x101000 /usr/lib64/libm-2.17.so
      0x7f7cdff71000     0x7f7ce00db000   0x16a000        0x0 /usr/local/gcc5.1/lib64/libstdc++.so.6.0.21
      0x7f7ce00db000     0x7f7ce02db000   0x200000   0x16a000 /usr/local/gcc5.1/lib64/libstdc++.so.6.0.21
      0x7f7ce02db000     0x7f7ce02e5000     0xa000   0x16a000 /usr/local/gcc5.1/lib64/libstdc++.so.6.0.21
      0x7f7ce02e5000     0x7f7ce02e7000     0x2000   0x174000 /usr/local/gcc5.1/lib64/libstdc++.so.6.0.21
      0x7f7ce02eb000     0x7f7ce030d000    0x22000        0x0 /usr/lib64/ld-2.17.so
      0x7f7ce050a000     0x7f7ce050b000     0x1000        0x0 /dev/zero (deleted)
      0x7f7ce050c000     0x7f7ce050d000     0x1000    0x21000 /usr/lib64/ld-2.17.so
```

图 7

1.2 多进程相关

1.2.1fork (vfork)

1.2.1.1 fork

- 采用了写时复制技术，copy on write。
- fork 调用之后产生两个进程，都从 fork 返回处进行，父进程中返回值为子进程 id，子进程中返回值为 0。
- 子进程与父进程共用代码段，但有各自的数据段、堆段和栈段。其中子进程的这三各段是父进程的完全拷贝，即这三段中的变量地址（虚拟地址）在父子进程中保持一致。

- fork 返回-1 失败原因，超过该用户或者该系统的最大可创建进程数限制。
- fork 之后，父子进程执行先后顺序不能确定，类比 vfork。
- 父子进程之间共享文件描述符，其中包含文件偏移量、文件状态等。文件描述符使用了引用计数技术。父子进程对文件的输入不会互相覆盖，但会出现顺序随意混杂，这里需要进程间同步。

1.2.1.2 vfork

- 子进程调用 exec 系列调用或者_exit 之前，将暂停父进程的执行，即使子进程 sleep。
- 子进程调用 exec 系列调用或者_exit 之前，子进程共享父进程的内存，也就是说子进程在子进程调用 exec 系列调用或者_exit 之前，对内存的操作对父进程可见。

1.3 进程间同步

1.3.1 概述

由于涉及到多进程间通信，所以需要支持进程间同步。

1.3.2 PTHRAD_PROCESS_SHARED

其支持互斥量 mutex 和条件变量 cond 在多进程场景下能够发挥相应作用。

设定方式为 pthread_mutexattr_setpshared / pthread_conattr_setpshared。

这里经测试验证，设置为 PTHRAD_PROCESS_SHARED 的条件变量和互斥量在同进程内多线程情况下也使用良好。即可用于多进程场景下任意进程内多线程之间的竞态关系。

1.3.3pthread::thread detached

注意对进程内不存在的句柄/或者对已 detached 的线程句柄做 detached 操作将会导致进程崩溃。

1.4 其他

1.4.1ftruncate

对打开的文件（包含虚拟文件以及磁盘文件）进行扩容，额外部分以'\0'填充。需要对应文件有写权限。

2.技术问题点

2.1new/delete 操作符替换

采用 gcc 下提供的属性设置方式__attribute__ 配合 alias 来实现。

Gcc 下分为强符号属性和弱符号属性，连接时若有多个同名的强符号就回报错。

编译器默认将函数和初始化了的全局变量作为强符号，而未初始化的全局变量为弱符号，也可以显示使用__attribute ((weak))来使用弱符号属性。

- 声明一个弱符号的函数，但不定义，连接可以通过。但调用该函数时将崩溃。
- 查看目标文件的符号类型命令：nm *.o
- 强符号、弱符号使用规则

  1）同名的强符号只能有一个，否则编译器会报重复定义。

  2）允许一个强符号和多个弱符号，但定义会选择强符号

  3）当有多个弱符号时，编译器会选择占用空间最大的那个。

对于使用了 c++11 的项目工程，new/delete 的版本相对有多种，包括可抛异常的和不可抛异常的版本，需要替换周全。

2.2 成员对象恢复（锁）

- 进程间不具有亲缘性，需要更新相关成员对象地址偏移修正（如该成员对象为指针类型，如非指针类型则不需要过多考虑）。
- 进程间具有亲缘性，不需要做地址偏移修正，得益于通过 fork 产生的新进程完整拷贝了老进程的堆段和栈段和数据段、BSS 段，即同一名称的变量其虚拟地址相同。

2.3 带父类的派生类恢复

其实需要的关注的点是虚表和虚表指针在进程内存布局中的存放位置。其中虚表在数据段中的静态区。2 而虚表指针位置是跟着对象走的，对象处在什么位置，那么虚表指针也将处在什么位置。

由上 2.2 节分析可知，新的进程会完整拷贝了老进程的堆段和栈段和数据段、BSS 段，也就是说会将基类的偏移量同样拷贝到新进程，言外之意是当派生类在新进程中恢复时不需要对其虚表指针做额外调整。

2.4 是否需要修正内存地址

分为两种场景，一种为不带亲缘性的多进程模型，另外一种是带亲缘性的多进程模型。

- 带亲缘性的多进程模型

借由 fork 系统调用的特性，完全可以不考虑共享内存地址偏移情况。

- 不带亲缘性的多进程模型

  由于不带有亲缘性，且每次 mmap 调用后返回的虚拟地址不同。需要考虑修正内存地址，方式为计算本次映射起始地址和上次映射起始地址的 offset（注意需要避免带符号和不带符号数值转换误差）。对后续使用的所有指针类型对象执行一次基于该 offset 的修正。

2.5 带亲缘性父子进程锁的继承

fork 产生的子进程会继承父进程锁的状态。如果锁不具有跨进程属性（posix mutex 不设置 POSIX_PROESS_SHARED 属性以及基于非共享内存创建该锁），就会导致死锁。

换而言之，对锁设置跨进程属性可以避免多进程场景下死锁问题。原因是锁的状态父子进程都可见（共享内存）。

3.整体设计思路

3.1 基础部分概述

如图 8，自上而下分为 4 层，分别是业务层、内存池层、共享内存衔接层和操作系统层。

其中操作系统层主要是共享内存操作 API 的选择，posix 系列和 systemV 系列的区别在上文 1.1.3 节已简单介绍。

共享内存衔接层作用为屏蔽操作系统层 API 操作接口差异性，管理底层共享内存分配以及提供恢复插槽。恢复插槽可以理解为业务索引，每一个索引（插槽）对应一部分业务功能。目前仅有内存池独占一个索引，实际其他业务不存在索引。由于时间限制，目前共享内存衔接层的 program-break 只增不减，后续考虑按页对共享内存进行管理，提升内存可复用性。

内存池层提供大块内存分配器和小块内存分配器，内存池中分配的内存都来自共享内存，而业务层使用的内存都来自内存池中。

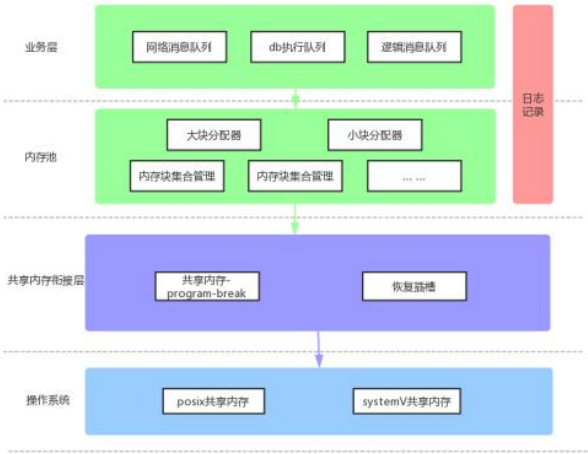

图 8

3.2 业务层部分

3.2.1 概述

由于业务层采用传统业务逻辑单线程的方式，所以可以将网络部分和业务部分做拆分。整个业务逻辑恢复存在两种设计方案。

- 方案 1

  将网络部分放在父进程，业务部分放在子进程中。如此，业务部分和网络部分的通信将是跨进程的，需要考虑进程间同步（见 1.3 节）。一旦业务部分（子进程）退出，则由父进程重新 fork 新业务进程。

- 方案 2

  将网络部分放在子进程 A，业务部分放在子进程 B 中。如此，业务部分和网络部分的通信同样页是跨进程的。一旦业务部分（子进程 B）退出，则由父进程重新 fork 新业务进程。父进程的唯一作用是“守护子进程，重新拉起子进程”。对于子进程 A（网络部分），由于逻辑足够简单，可以视其不会退出。

3.2.2 方案比较

方案 2 的优势在于父子之间的继承的内容将是清晰可控的。劣势是将会产生起码 3 个同名进程，增加了调试复杂度。

方案 1 的优势在于在拆分网络跟业务之后，仅额外多增加了一个新同名进程。劣势是当业务子进程退出被重新拉起后，新的业务进程会继承父进程网络相关部分的内容，存在安全隐患。

3.2.3 注意点

不管是方案 1 还是方案 2，都需要将需要恢复的所有对象在 fork 之前初始化完毕（需借由基于共享内存的内存池分配内存）。原因是在 fork 之后，子进程会拷贝父进程当前堆栈中的变量内容（见 1.2 节）。如此时不预先初始化，则其值为空，子进程被拉起后一旦用到了该变量将会为其重新分配内存达不到多进程操作同一份内存区域要求。

4.补充

4.1 一些特别的点

- Win10 下带的 linux 子系统在使用 shmget 返回的 shmID 一直为 0。

4.2 接入 shm 基础库搭建恢复服务注意点

- 将服务逻辑中所有需要恢复的数据管理器的构建操作放在 fork 前。目的是将此类管理器对象在共享内存上构建，并让子进程继承，一般此类管理器都以单例形式提供。换而言之，子进程继承的是指向对应共享内存的指针（绳子）。
- 由于 fork 后，子进程只复制父进程的一个线程，所以如果管理器中需要开额外线程处理则需要在 fork 后的子进程中重新开启额外线程处理。注意如果父子进程都有相应处理线程就需要考虑跨进程锁。
- 需要将全局的 new/delete 替换为 shm 版本的 new/delete。虽说全局替换会有隐患，隐患集中在使用第三方库时其分配方式不可知。但从兼容 stl 容器角度来说，省了大量时间。
- shm 分配器必须在最开始初始化完毕。需要恢复的业务代码中不可出现 malloc 以及 free。