ltp 问题分析之 madvise

Cauchy Pu

1 问题概述

夹杂各种事情，终于 ltp 的 madvise 的分析报告出炉，自本次报告后我的工作重心可能就要转移了。

先看问题现象，amdvise 共有几组测试例程，目前大部分都有问题。

```
sudo ./madvise01
CONF: MADV_HWPOISON is not supported
CONF: MADV_MERGEABLE is not supported
CONF: MADV_UNMERGEABLE is not supported
CONF: MADV_HUGEPAGE is not supported
CONF: MADV_NOHUGEPAGE is not supported
```

```
sudo ./madvise02
CONF: MADV_WILLNEED is not supported
```

```
sudo ./madvise06
CONF: memory cgroup needed
```

```
sudo ./madvise07
CONF: CONFIG_MEMORY_FAILURE probably not set in kconfig: EINVAL
```

```
sudo ./madvise09
CONF: '/sys/fs/cgroup/memory/' not present, CONFIG_MEMCG missing?
```

2 问题分析

madvise 是啥呢？先有个基础认识，摘抄一段 man madvise：

> int madvise(void *addr, size_t length, int advice):
>
> The madvise() system call is used to give advice or directions to the kernel about the address range beginning at address addr and with size length bytes. In most cases, the goal of such advice is to improve system or application performance.

看起来用途蛮大，用来给内核建议一段内存该怎么用，可以用来改善系统或应用程序的性能。但是申威平台上报出了很多不支持的错误，根据前述经验，“不支持”可能不是真的“不支持”，还得仔细分析内核配置、内核代码以及做实验验证每一种具体情况。

只有一个一个分析上述错误了。

1. madvise01

   这些 MADV 开头的其实都是 advice 的一种，下面来具体看看每种是什么意义。

   (a) MADV_HWPOISON
   首先是 man madvise 的信息：

> Poison the pages in the range specified by addr and length and handle subsequent references to those pages like a hardware memory corruption. This operation is available only for privileged (CAP_SYS_ADMIN) processes. This operation may result in the calling process receiving a SIGBUS and the page being unmapped.
>
> This feature is intended for testing of memory error-handling code; it is available only if the kernel was configured with CONFIG_MEMORY_FAILURE.

概要地说，这是一个 feature，需要打开 CONFIG_MEMORY_FAILURE 配置。理论上，这个 feature 并没有太多的用处，只是把一个地址范围标记为 poison，然后对这块地址的访问就像是在访问一块“内存出错”了的地址，这个 feature 可以用来测试“内存出错处理代码”。

那么，什么叫“内存出错”，以及“内存出错处理代码”长啥样？暂且不表。

现在尝试去打开 MEMORY_FAILURE，当然不是直接就打开这个内核选项的。帮助信息还是要看看：

> Enables code to recover from some memory failures on systems with MCA recovery. This allows a system to continue running even when some of its memory has uncorrected errors. This requires special hardware support and typically ECC memory.

有几个专业名词了，MCA recovery 是啥？ECC memory 是啥？稍微 Google 下，参考文献 1、2 给出了点有用信息。简单说就是，ECC memory 是一种特别的内存，它可以在内存受到辐射等情况下时，出现了 bit 翻转也能纠错从而继续工作。为了配合 ECC memory 的这种特点，我们 OS 开发人员当然就有工作啦，比如现在 ECC memory 出错了，我们的内核是不是要提供个处理函数啊，这个就叫 MCA recovery。

那这跟 MADV_HWPOISON 标志有什么关系嘛。

嗯，其实就是，ECC memory 出错（比如本来存的是比特 0，后来翻转为 1）概率其实是很小的，这主要在金融以及数据中心可能会考虑这种情况，或者是，你的计算机系统要运行在外太空，辐射较强，翻转机率提高。

那我们写在内核里的“memory error-handling code”的代码就没法测了呀，咱也不能带着我们的 Linux 飞到外太空去不是，即使能这样做，在外太空翻转的概率也只是提高了，而不是一定。

所以内核开发人员弄了个 MEMORY_FAILURE 的 feature，给某段内存标上这个标志后，访问这段内存就是“like a hardware memory corruption”，像是访问内存出错的地方。这样就可以触发我们的“memory error-handlingcode”了。

说来说去，这个 feature 是为模拟测试硬件准备的，就是我们现在没有 ECC memory 这种“高级货”的时候也能写并测试“memory error-handling code”，暂时就先到这里吧。

接下来先打开 MEMORY_FAILURE 编译下。不出意外编译会有错，没错才是不正常。mm/madvise.c 会报出有一些宏常数定义找不到，于是找到定义这些宏的文件，文件 usr/include/asm-generic/mman-common.h 即是。将 mman-common.h 拷贝一份到 include/linux 下，同时修改 mm/madvise.c 加上头文件 #include <linux/mman-common.h>，再编译，又会有另外的宏重复定义，在 include/linux/mman-common.h 中注释这些宏，于是新内核就可以编译出来测试了，发现本项测试通过。暂且到这儿吧，后面 madvise07 其实还有有问题，到那时再表。最后要注意，这种修改其实是不“正规的”，不过为了往下测试验证也未尝不可，到用户真的有需要 CONFIG_MEMORY_FAILURE，我们再和研究院沟通或他们做，或我们自己“正规”的来把这事顺手做了。

(b) MADV_MERGEABLE
首先还是 man madvise 的信息了：

> Enable Kernel Samepage Merging (KSM) for the pages in the range specified by addr and length. The kernel regularly scans those areas of user memory that have been marked as mergeable, looking for pages with identical content. These are replaced by a single write-protected page (which is automatically copied if a process later wants to update the contentof the page).
>
> The MADV_MERGEABLE and MADV_UNMERGEABLE operations are available only if the kernel was configured with CONFIG_KSM.

就是建议内核把相同的内容的 page merge 起来以节省内存。再结合参考信息 3，此 feature 不涉及硬件信息，纯属打开内核配置的问题，经测试，打开 CONFIG_KSM 配置后测试通过。

(c) MADV_UNMERGEABLE
与 MADV_MERGEABLE 是配对的。

(d) MADV_HUGEPAGE
本项依旧是软件层面的事情，与硬件无关，内核社区自己已实现，打开内核配置 CONFIG_TRANSPARENT_HUGEPAGE 即可通过测试，与上述 MADV_MERGEABLE 类似，不再赘述。

(e) MADV_NOHUGEPAGE
与 MADV_HUGEPAGE 是配对的。

2. madvise02

(a) MADV_WILLNEED
看看 man 信息：

> Expect access in the near future. (Hence, it might be a good idea to read some pages ahead.)

把一个页面标记为 MADV_WILLNEED 会建议内核提前把页面准备好，方便后面使用提高效率。这个特点没有要求任何内核特殊的配置。按道理不会打印 is not supported 啊。

于是检查测试代码 madvise02.c，这里打印了串：

```
if (tc->skip == 1) {
 tst_res(TCONF, "%s is not supported", tc->name);
 return;
}
```

而 tc->skip 在这里被改变成 1：

```
if ((tst_kvercmp(3, 9, 0)) > 0 &&
                                tc->exp_errno == EBADF)
 tc->skip = 1;
```

经查 tst_kvercmp 会比较内核版本，我们的版本是 4.19 大于 3.9，EBADF 是说映射已经存在但不是一个文件。所有这一切均不是在确认 madvise 函数是否支持 MADV_WILLNEED，故认为这个 not supported 打印具有误导性，可以和社区沟通它究竟想表达什么。后面的实验验证会说明实际上 MADV_WILLNEED 是支持的。

另外值得一提的是，为了修复 madvise01.c 的 MADV_MERGEABLE is not supported 问题，打开内核 CONFIG_KSM 选项，发现 madvise02 出现了 MADV_MERGEABLE 和 MADV_UNMERGEABLE is not supported 打印。

实际上可以肯定，来自参考文献和内核代码实现的佐证，打开 CONFIG_KSM，madvise 函数的 MERGEABLE 功能就会开起，这在前述 madvise01 分析中已经说明。

那么为什么还会有这个打印呢? 和 MADV_WILLNEED 打印出来的流程类似，只不过这次是 madvise02.c 的下面代码修改了 tc->skip：

```
if (access(KSM_SYS_DIR, F_OK) == 0)
 tc->skip = 1;
```

KSM_SYS_DIR 被定义为/sys/kernel/mm/ksm，这在内核配置为 CONFIG_KSM 时该文件自然存在。而 F_OK 是测试文件的存在性，故条件满足。tc->skip 被设置为 1，进而 not supported 被打印，同样的情况，这个打印具有误导性，因为内核已经被配置为 CONFIG_KSM。

3. madvise06
打印是 “memory cgroup needed”，看样子是差什么东西，memory 和 cgroup，那差的这个是啥东西啊。

看代码是这里打印出来的：

```
#define MNT_NAME "memory"
SAFE_MKDIR(MNT_NAME, 0700);
if (mount("memory", MNT_NAME, "cgroup", 0, "memory") == -1) {
 if (errno == ENODEV || errno == ENOENT)
  tst_brk(TCONF, "memory cgroup needed");
}
```

看样子是要看看 mount 了，老朋友 man 这么说：

> int mount(const char *source, const char *target, const char *filesystemtype, unsigned long mountflags, const void *data):
> mount() attaches the filesystem specified by source (which is often a pathname referring to a device, but can also be the pathname of a directory or file, or a dummy string) to the location (a directory or file) specified by the pathname in target.
>
> The data argument is interpreted by the different filesystems. Typically it is a string of comma-separated options understood by this filesystem. See mount(8) for details of the options available foreach filesystem type.

其中要注意的第二段,我已经故意摘出来了。我们要挂载的文件系统是 cgroup,那么这个 filesystem 是怎么解释它的 data 的呢?

简言之，/sys/fs/cgroup 下有许多 controller，而我们关心的是那个叫 memory(就是 data 参数指明的) 的 controller，把它挂到由 MNT_NAME 指明的路径下... 等等，啥叫 cgroup，controller 又是啥啊？这个要看看参考文献 4、5 了。

ls /sys/fs/cgroup/memory 看下，嗯，确实没有这个目录，那它该怎么来呢?

参考文献文献 5 讲了下，要打开 CONFIG_MEMCG 配置，如此一试，果真如此。报错没有了，madvise06 测试也过了。当然个中细节还要细看参考文献 4、5 啊。

4. madvise07

madvise07 要费点劲了。

首先是没有前述的配置选项打开时,错误表现为前述问题概述节的 madvise07 贴图。打开 CONFIG_MEMORY_FAILURE 时，错误表现为：

```
FAIL: Did not receive SIGBUS on accessing poisoned page
```

怎么就收不到 SIGBUS 信号了呢? 本情况下, 多半是申威内核差点东西, 因为前述提到 MEMORY_FAILURE 需要特殊的内存类型, 但是, 这么点“东西”是啥呢? 看来还需要细细探究一番。

申威内核运行 madvise07 是上诉 did not receive 的表现, 那 Intel 内核呢? 于是在 X86-64 平台运行了下 madvise07, 结果正常, 打印:

```
PASS: Received SIGBUS after accessing poisoned page
```

这说明 X86-64 上可以收到 SIGBUS 信号。那就看看 madvise07.c 了, 关键的是这段 (重组了下顺序):

```
//函数 run_child
if (madvise(mem, msize, MADV_HWPOISON) == -1) {
 if (errno == EINVAL) {
  tst_res(TCONF | TERRNO,
  "CONFIG_MEMORY_FAILURE probably not set in kconfig");
   } else {
      tst_res(TFAIL | TERRNO, "Could not poison memory");
   }
  exit(0);
}

*((char *)mem) = 'd';
tst_res(TFAIL, "Did not receive SIGBUS on accessing poisoned page");

//另一个函数 run
if (pid == 0) {
 run_child();
 exit(0);
}
SAFE_WAITPID(pid, &status, 0);
        if (WIFSIGNALED(status) && WTERMSIG(status) == SIGBUS) {
         tst_res(TPASS, "Received SIGBUS after accessing poisoned page");
     return;
}
```

简单来说就是, 父进程起了个子进程运行 run_child 函数, 而在 run_child 函数里, 调用了 madvise 把一段内存标记为 MADV_HWPOISON, 随后语句 *((char *)mem) = 'd' 访问了这段内存。按道理, 访问一段标记为 MADV_HWPOISON 的内存应该被发送 SIGBUS 信号, 从而子进程被杀死, 父进程里打印出 Received SIGBUS after accessing poisoned page。如果被杀死的子进程没有收到 SIGBUS, 也就是语句 *((char *)mem) = 'd' 访问并没有引起子进程异常退出, 所以后面的一句:

```
tst_res(TFAIL, "Did not receive SIGBUS on accessing poisoned page");
```

会被正常执行, 就是在申威平台看到的这句话了, 它不会出现在 X86-64 平台。

userland 的原因如此, 还是那个问题申威内核究竟差了什么, 导致没能发送 SIGBUS 信号给用户进程?

参考文献 1 说了句蛮有用的话:

> The code consists of a the high level handler in mm/memory-failure.c, a new page poison bit and various checks in the VM to handle poisoned pages.

那就稍微研究下 mm/memory-failure.c 咯。

可以看到，函数 memory_failre 正是内存出错时会被调用的处理函数。它向下会一步步调用到 kill_proc 函数，而函数 kill_proc 里会调用 force_sig_mceerr 或 send_sig_mceerr 函数给应用程序发送信号并杀死应用进程。那我们就来看看内核里这两个函数的实现，它们实现在 signal.c 里：

```
int force_sig_mceerr(int code, void __user *addr, short lsb, struct
↪  task_struct *t)
{
        struct siginfo info;

        WARN_ON((code != BUS_MCEERR_AO) && (code != BUS_MCEERR_AR));
        clear_siginfo(&info);
        info.si_signo = SIGBUS;
        info.si_errno = 0;
        info.si_code = code;
        info.si_addr = addr;
        info.si_addr_lsb = lsb;
        return force_sig_info(info.si_signo, &info, t);
}

int send_sig_mceerr(int code, void __user *addr, short lsb, struct
↪  task_struct *t)
{
        struct siginfo info;

        WARN_ON((code != BUS_MCEERR_AO) && (code != BUS_MCEERR_AR));
        clear_siginfo(&info);
        info.si_signo = SIGBUS;
        info.si_errno = 0;
        info.si_code = code;
        info.si_addr = addr;
        info.si_addr_lsb = lsb;
        return send_sig_info(info.si_signo, &info, t);
}
```

终于看到心心念念的 SIGBUS 啊！就是这里确定给用户程序发送信号的。怎么验证这点呢？很简单啦，把它改成 SIGSEGV 这样的，再在 X86-64 平台运行 madvise07，打印就变成了 FAIL: Child killed by SIGSEGV，这正是因为我们精准改动要的效果哦。

而在申威平台这些都有，但还是没有 PASS 打印，那就是看看调用了 memory_failure 啦。这里就很关键了，是文件 arch/x86/kernel/cpu/mce/core.c！这个文件 arch/sw_64/下是没有的。

上述文件里的函数 do_machine_check 会调用 memory_failure。而这个 do_machine_check 函数实际是内存出错时的异常处理函数，异常产生时，会自动触发执行。至于异常产生怎么就自动执行函数了，说实话这对于内核或者现代 IT 基础来说很关键，不过在本文里并不关心，参考文献 6 给出了点信息，感兴趣的读者可以看看。

总之，到这里真相大白了，申威内核缺少响应内存错误的句柄函数 do_machine_check，如果用户有需要（幻想申威也能有这样的客户啊，搭载申威 CPU 飞向外太空？超级数据中心也用 SW CPU，国际金融中心相中太湖旁的 WIAT），我们就可以随手做了，注意这并不依赖要 ECC memory 才能做，正如前文所料，这正是 MEMORY_FAILURE 的存在意义。

5. madvise09

   根据提示首先是打开 CONFIG_MEMCG 配置了，错误转变成了：

   ```
   FAIL: Found corrupted page
   ```

   被标记为 MADV_FREE 的页，并不会立即释放，只有在遭遇内存压力时才会。而在标记为 MADV_FREE 和真正释放时之间有页又再次被释放的话，这些被再次写的页不应该被释放，4.19.90 的 Intel 内核遵循这个，但是申威 4.19.90 内核不遵循，这些被再次写过的页释放后填充 0，与第二次写入的值不一样，所以认为是 corrupted 了。

3 实验验证

分析中已经指明了如何验证，包括打开相关配置，查找、修改内核代码来验证。

4 解决方案

1. MADV_HWPOISON：配置 MEMORY_FAILURE 打开，同时实现 Intel 下等价的异常函数 do_machine_check。
2. MADV_MERGEABLE：配置 CONFIG_KSM 打开。
3. MADV_HUGEPAGE：配置 TRANSPARENT_HUGEPAGE 打开。
4. MADV_WILLNEED：跳过测试，与社区沟通 not supported 打印误导。
5. 打开 CONFIG_MEMCG 来支持 cgroup 的 memory controller。
6. 以 MADV_FREE 标志调用 madvise 时，注意再次被写的页也会被释放，终极解决方案需要修正 cgroup 的 memory controller 实现以使得符合标准。

5 参考文献

1. https://www.kernel.org/doc/html/v4.18/vm/hwpoison.html
2. https://en.wikipedia.org/wiki/ECC_memory
3. https://www.kernel.org/doc/html/v4.19/admin-guide/mm/ksm.html
4. https://man7.org/linux/man-pages/man7/cgroups.7.html
5. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt
6. https://www.kernel.org/doc/html/latest/x86/exception-tables.html