



下载APP



## 07 案例篇 | 如何预防内存泄漏导致的系统假死?

2020-09-03 邵亚方

Linux内核技术实战课

[进入课程 >](#)**讲述：邵亚方**

时长 12:00 大小 11.00M



你好，我是邵亚方。

上节课，我们讲了有哪些进程的内存类型会容易引起内存泄漏，这一讲我们来聊一聊，到底应该如何应对内存泄漏的问题。

我们知道，内存泄漏是件非常容易发生的事，但如果它不会给应用程序和系统造成危害，那它就不会构成威胁。当然我不是说这类内存泄漏无需去关心，对追求完美的程序员而言，还是需要彻底地解决掉它的。



而有一些内存泄漏你却需要格外重视，比如说长期运行的后台进程的内存泄漏，这种泄漏日积月累，会逐渐耗光系统内存，甚至会引起系统假死。

我们在了解内存泄漏造成的危害之前，先一起看下什么样的内存泄漏是有危害的。

## 什么样的内存泄漏是有危害的？

下面是一个内存泄漏的简单示例程序。

[复制代码](#)

```
1 #include <stdlib.h>
2 #include <string.h>
3
4 #define SIZE (1024 * 1024 * 1024) /* 1G */
5 int main()
6 {
7     char *p = malloc(SIZE);
8     if (!p)
9         return -1;
10
11     memset(p, 1, SIZE);
12     /* 然后就再也不使用这块内存空间 */
13     /* 没有释放p所指向的内存进程就退出了 */
14     /* free(p); */
15     return 0;
16 }
```

我们可以看到，这个程序里面申请了 1G 的内存后，没有进行释放就退出了，那这 1G 的内存空间是泄漏了吗？

我们可以使用一个简单的内存泄漏检查工具 (valgrind) 来看看。

[复制代码](#)

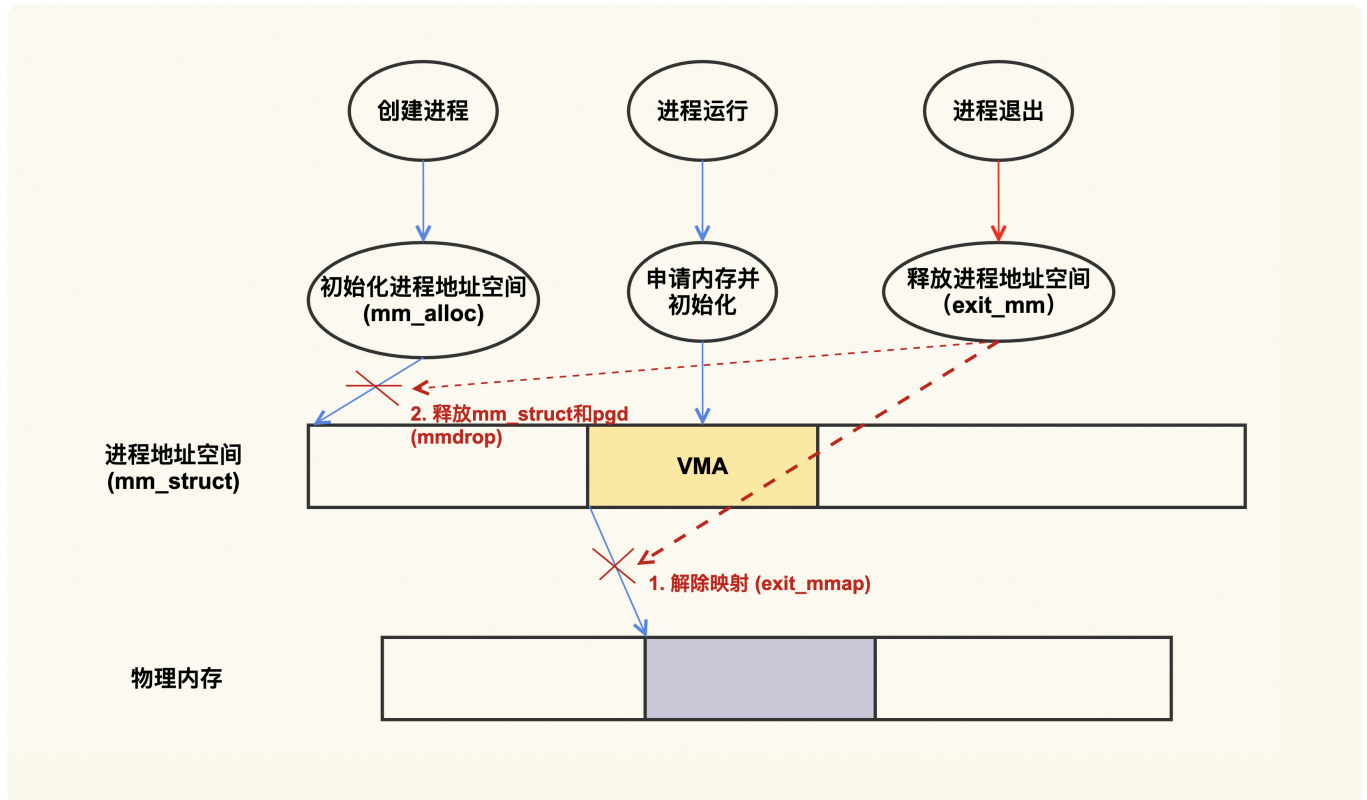
```
1 $ valgrind --leak-check=full ./a.out
2 ==20146== HEAP SUMMARY:
3 ==20146==      in use at exit: 1,073,741,824 bytes in 1 blocks
4 ==20146==    total heap usage: 1 allocs, 0 frees, 1,073,741,824 bytes allocated
5 ==20146==
6 ==20146== 1,073,741,824 bytes in 1 blocks are possibly lost in loss record 1 o
7 ==20146==    at 0x4C29F73: malloc (vg_replace_malloc.c:309)
8 ==20146==    by 0x400543: main (in /home/yafang/test/mmleak/a.out)
9 ==20146==
10 ==20146== LEAK SUMMARY:
11 ==20146==    definitely lost: 0 bytes in 0 blocks
12 ==20146==    indirectly lost: 0 bytes in 0 blocks
13 ==20146==    possibly lost: 1,073,741,824 bytes in 1 blocks
14 ==20146==    still reachable: 0 bytes in 0 blocks
```

```
15 ==20146==
```

```
suppressed: 0 bytes in 0 blocks
```

从 valgrind 的检查结果里我们可以清楚地看到，申请的内存只被使用了一次（memset）就再没被使用，但是在使用完后却没有把这段内存空间给释放掉，这就是典型的内存泄漏。那这个内存泄漏是有危害的吗？

这就要从进程地址空间的分配和销毁来说起，下面是一个简单的示意图：



进程地址空间申请和释放示意图

从上图可以看出，进程在退出的时候，会把它建立的映射都给解除掉。换句话说，进程退出时，会把它申请的内存都给释放掉，这个内存泄漏就是没危害的。不过话说回来，虽然这样没有什么危害，但是我们最好还是要在程序里加上 `free $`，这才是符合编程规范的。我们修改一下这个程序，加上 `free $`，再次编译后通过 valgrind 来检查，就会发现不存在任何内存泄漏了：

复制代码

```
1 $ valgrind --leak-check=full ./a.out
2 ==20123== HEAP SUMMARY:
3 ==20123==    in use at exit: 0 bytes in 0 blocks
4 ==20123== total heap usage: 1 allocs, 1 frees, 1,073,741,824 bytes allocated
5 ==20123==
6 ==20123== All heap blocks were freed -- no leaks are possible
```

总之，如果进程不是长时间运行，那么即使存在内存泄漏（比如这个例子中的只有 malloc 没有 free），它的危害也不大，因为进程退出时，内核会把进程申请的内存都给释放掉。

我们前面举的这个例子是对应用程序无害的内存泄漏，我们继续来看下哪些内存泄漏会给应用程序产生危害。我们同样以 malloc 为例，看一个简单的示例程序：

 复制代码

```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <unistd.h>
4
5  #define SIZE (1024 * 1024 * 1024) /* 1G */
6
7  void process_memory()
8  {
9      char *p;
10     p = malloc(SIZE);
11     if (!p)
12         return;
13     memset(p, 1, SIZE);
14     /* Forget to free this memory */
15 }
16
17 /* 处理其他事务，为了简便起见，我们就以sleep为例 */
18 void process_others()
19 {
20     sleep(1);
21 }
22
23
24 int main()
25 {
26     /* 这部分内存只处理一次，以后再也不用用到 */
27     process_memory();
28
29
30     /* 进程会长时间运行 */
31     while (1) {
32         process_others();
33     }
34     return 0;
```


这是一个长时间运行的程序，process\_memory() 中我们申请了 1G 的内存去使用，然后就再也不用它了，由于这部分内存不会再被利用，这就造成了内存的浪费，如果这样的程

序多了，被泄漏出去的内存就会越来越多，然后系统中的可用内存就会越来越少。

对于后台服务型的业务而言，基本上都是需要长时间运行的程序，所以后台服务的内存泄漏会给系统造成实际的危害。那么，究竟会带来什么样的危害，我们又该如何去应对呢？

## 如何预防内存泄漏导致的危害？

我们还是以上面这个 `malloc()` 程序为例，在这个例子中，它只是申请了 1G 的内存，如果说持续不断地申请内存而不释放，你会发现，很快系统内存就会被耗尽，进而触发 OOM killer 去杀进程。这个信息可以通过 `dmesg`（该命令是用来查看内核日志的）这个命令来查看：

 复制代码

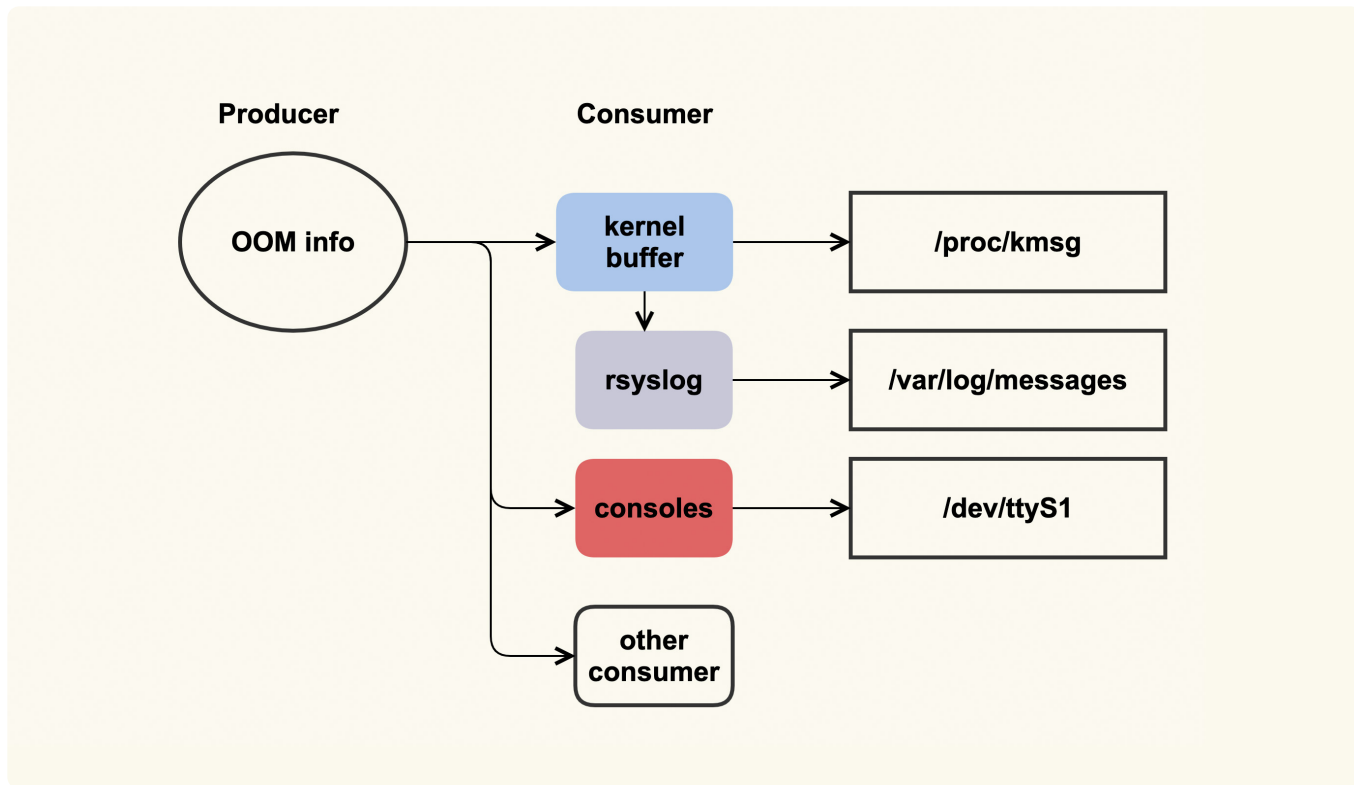
```
1 $ dmesg
2 [944835.029319] a.out invoked oom-killer: gfp_mask=0x100dca(GFP_HIGHUSER_MOVAB
3 [...]
4 [944835.052448] Out of memory: Killed process 1426 (a.out) total-vm:8392864kB,
```

系统内存不足时会唤醒 OOM killer 来选择一个进程给杀掉，在我们这个例子中它杀掉了这个正在内存泄漏的程序，该进程被杀掉后，整个系统也就变得安全了。但是你要注意，**OOM killer 选择进程是有策略的，它未必一定会杀掉正在内存泄漏的进程，很有可能是一个无辜的进程被杀掉。**而且，OOM 本身也会带来一些副作用。

我来说一个发生在生产环境中的实际案例，这个案例我也曾经 [反馈给 Linux 内核社区来做改进](#)，接下来我们详细说一下它。

这个案例跟 OOM 日志有关，OOM 日志可以理解为是一个单生产者多消费者的模型，如下图所示：

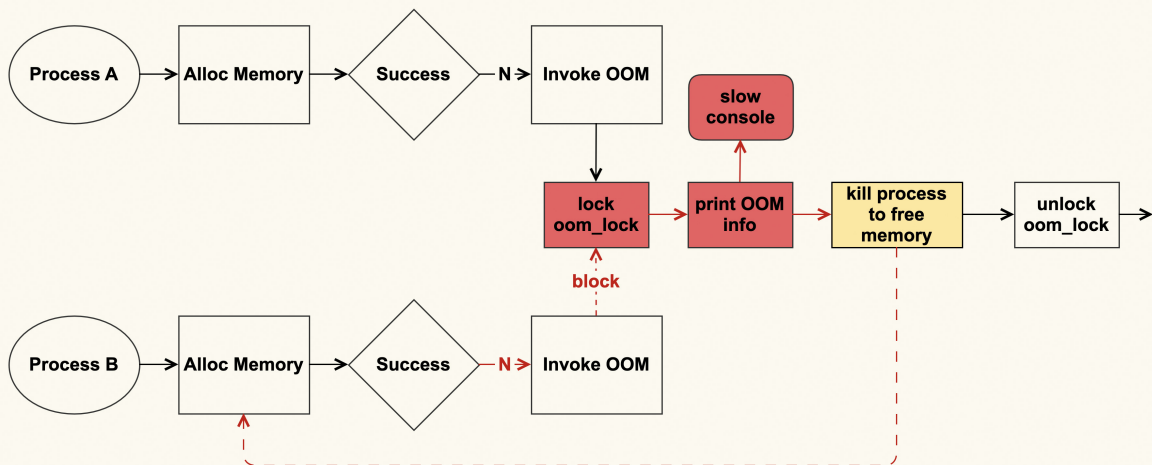




OOM info

这个单生产者多消费者模型，其实是由 OOM killer 打印日志（OOM info）时所使用的 printk（类似于 userspace 的 printf）机制来决定的。printk 会检查这些日志需要输出给哪些消费者，比如写入到内核缓冲区（kernel buffer），然后通过 dmesg 命令来查看；我们通常也都会配置 rsyslog，然后 rsyslogd 会将内核缓冲区的内容给转储到日志文件（/var/log/messages）中；服务器也可能会连着一些控制台（console），比如串口，这些日志也会输出到这些 console。

问题就出在 console 这里，如果 console 的速率很慢，输出太多日志会非常消耗时间，而当时我们配置了 “console=ttyS1,19200”，即波特率为 19200 的串口，这是个很低速率的串口。一个完整的 OOM info 需要约 10s 才能打印完，这在系统内存紧张时就会成为一个瓶颈点，为什么会是瓶颈点呢？答案如下图所示：



OOM为什么会成为瓶颈点

进程 A 在申请内存失败后会触发 OOM，在发生 OOM 的时候会打印很多很多日志（这些日志是为了方便分析为什么 OOM 会发生），然后会选择一个合适的进程来杀掉，从而释放出来空闲的内存，这些空闲的内存就可以满足后续内存申请了。

如果这个 OOM 的过程耗时很长（即打印到 slow console 所需的时间太长，如上图红色部分所示），其他进程（进程 B）也在此时申请内存，也会申请失败，于是进程 B 同样也会触发 OOM 来尝试释放内存，而 OOM 这里又有一个全局锁（oom\_lock）来进行保护，进程 B 尝试获取（trylock）这个锁的时候会失败，就只能再次重试。

如果此时系统中有很多进程都在申请内存，那么这些申请内存的进程都会被阻塞在这里，这就形成了一个恶性循环，甚至会引发系统长时间无响应（假死）。

针对这个问题，我与 Linux 内核内存子系统的维护者 Michal Hocko 以及 OOM 子模块的活跃开发者 Tetsuo Handa 进行了 [一些讨论](#)，不过我们并没有讨论出一个完美的解决方案，目前仍然是只有一些规避措施，如下：

### 在发生 OOM 时尽可能少地打印信息

通过将 [vm.oom\\_dump\\_tasks](#) 调整为 0，可以不去备份（dump）当前系统中所有可被 kill 的进程信息，如果系统中有很多进程，这些信息的打印可能会非常消耗时间。在我们这个案例里，这部分耗时约为 6s 多，占 OOM 整体耗时 10s 的一多半，所以减少这部分的打印能够缓解这个问题。

但是，**这并不是一个完美的方案，只是一个规避措施**。因为当我们把 `vm.oom_dump_tasks` 配置为 1 时，是可以通过这些打印的信息来检查 OOM killer 是否选择了合理的进程，以及系统中是否存在不合理的 OOM 配置策略的。如果我们将它配置为 0，就无法得到这些信息了，而且这些信息不仅不会打印到串口，也不会打印到内核缓冲区，导致无法被转储到不会产生问题的日志文件中。

### 调整串口打印级别，不将 OOM 信息打印到串口

通过调整 `/proc/sys/kernel/printk` 可以做到避免将 OOM 信息输出到串口，我们通过设置 `console_loglevel` 来将它的级别设置的比 OOM 日志级别（为 4）小，就可以避免 OOM 的信息打印到 console，比如将它设置为 3：

 复制代码

```
1 # 初始配置(为7)：所有信息都会输出到console
2 $ cat /proc/sys/kernel/printk
3 7 4 1 7
4
5 # 调整console_loglevel级别，不让OOM信息打印到console
6 $ echo "3 4 1 7" > /proc/sys/kernel/printk
7
8 # 查看调整后的配置
9 $ cat /proc/sys/kernel/printk
10 3 4 1
```

但是这样做会导致所有低于默认级别（为 4）的内核日志都无法输出到 console，在系统出现问题时，我们有时候（比如无法登录到服务器上面时）会需要查看 console 信息来判断问题是什么引起的，如果某些信息没有被打印到 console，可能会影响我们的分析。

这两种规避方案各有利弊，你需要根据你的实际情况来做选择，如果你不清楚怎么选择时，我建议你选择第二种，因为我们使用 console 的概率还是较少一些，所以第二种方案的影响也相对较小一些。

OOM 相关的一些日志输出后，就到了下一个阶段：选择一个最需要杀死的进程来杀掉。OOM killer 在选择杀掉哪个进程时，也是一个比较复杂的过程，而且如果配置不当也会引起其他问题。关于这部分的案例，我们会在下节课来分析。

## 课堂总结



这节课我们讲了什么是内存泄漏，以及内存泄漏可能造成的危害。对于长时间运行的后台任务而言，它存在的内存泄漏可能会给系统带来比较严重的危害，所以我们一定要重视这些任务的内存泄漏问题。

内存泄漏问题是非常容易发生的，所以我们需要提前做好内存泄漏的兜底工作：即使有泄漏了也不要让它给系统带来很大的危害。长时间的内存泄漏问题最后基本都会以 OOM 结束，所以你需要去掌握 OOM 的相关知识，来做好这个兜底工作。

如果你的服务器有慢速的串口设备，那你一定要防止它接收太多的日志，尤其是 OOM 产生的日志，因为 OOM 的日志量是很大的，打印完整个 OOM 信息 kernel 会很耗时，进而导致阻塞申请内存的进程，甚至会严重到让整个系统假死。

墨菲定律告诉我们，如果事情有变坏的可能，不管这种可能性有多小，它总会发生。对应到内存泄漏就是，当你的系统足够复杂后，它总是可能会发生的。所以，对于内存泄漏问题，你在做好预防的同时，也一定要对它发生后可能带来的危害做好预防。

## 课后作业

请写一些应用程序来构造内存泄漏的测试用例，然后使用 valgrind 来进行观察。欢迎在留言区分享你的看法。

感谢你的阅读，如果你认为这节课的内容有收获，也欢迎把它分享给你的朋友，我们下一讲见。

提建议

## 更多课程推荐

# 程序员的数学基础课

在实战中重新理解数学

黄申

LinkedIn 资深数据科学家



涨价倒计时 🕒

今日秒杀 **¥79**, 9月11日涨价至 **¥129**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 06 基础篇 | 进程的哪些内存类型容易引起内存泄漏？

下一篇 08 案例篇 | Shmem：进程没有消耗内存，内存哪去了？

## 精选留言 (4)

写留言



不倒翁

2020-09-03

提个建议：案例这方面，教师能否改变一下讲课顺序

运行一个有问题的程序(不看源码)-->观察系统指标发现可能是内存泄漏 -->用工具找到问题程序 -->分析可能是哪方面的问题-->查看源码 -->问题背后的原理等

作者回复: 谢谢你的建议。这样我也考虑过，只是我们感觉这有点像应试教育先出题再说答案，我主要是担心这会局限住大家的思维。所以我在案例篇里一般都是把具体问题给做一个抽象，能够让大家不局限于某一个问题，而是通过这些抽象的案例来理解问题的本质。



3

**大飞哥**

2020-09-03

太棒了，谢谢老师！最近项目也出现过假死的现象，lockdep、soft lock等，现在还没找到好的方法去查找和定位，望老师指教。之前也出现过打印日志拉慢系统性能的情况，后面也发现是printk的原因，后面修改printk不从串口输出，打印到网络就解决了，当时也没时间追溯根本原因，现在一想，printk里面有几把锁，是否也是频繁和大数据打印出现锁竞争所导致的？项目需要用到很多特殊硬件资源和DMA，所以大多是内核层开发，成...  
展开 ✓

作者回复: 在出现这种假死问题时需要首先想办法来保存事故现场，有了这些现场后就可以分析具体原因了，lockdep和softlockup这些都可以认为是内核问题，所以最好可以构造出来vmcore来进行分析，我通常使用sysrq来构造出来vmcore，然后分析这些vmcore。这里有个问题时，往往发生问题的时候就没有办法来执行命令了，所以根据一些信息来提前做好预判很重要，比如根据load值达到多少就认为异常了，或者利用其他一些系统指标，比如内存引起的假死可以借助psi指标来做预判。在构造vmcore时，如果是磁盘故障或者文件系统出了问题，那这个vmcore也可能无法保存下来，这个时候借助网络把这些信息给导出来也是一个方案，嵌入式系统中常用的做法这是把他们保存在非易失内存中，然后重启后解析这些非易失内存。



💬 1

👍 2

**jssfy**

2020-09-03

如果宕机的话kernel buffer的数据可能没来得及落盘，这个一般怎么解？

作者回复: 异常宕机是总会发生的，如果这个时候有未落盘的数据，那这部分数据就会丢失，这其实是难以去避免的。所以这种问题大致是有两个思路：1.数据没保存完整的话，已保存的数据是准确的就好，文件系统会做校验，不完整的数据认为是无效数据。2. 冗余备份来预防单点故障，一台主机宕机很正常，如果多台同时宕机就几乎不可能了，所以冗余备份是很重要的。



💬

👍 2

**张振宇**

2020-09-07

老师，磁盘经常会有文件权限变成??? 的情况，使用xfs repair可以修复，这个是业务的问题吗

作者回复: xfs存在问题的可能性更大



