

OSPP 2022



项目申请书



项目编号: 22b970497

利用 bpf 统计内核中 alloc_page 内存信息

项目主导师 : 刘勇强
主导师邮箱 : liuyongqiang13@huawei.com
申请人 : 龚宇晨
申请人邮箱 : gyc8787@163.com
申请日期 : 2022.5.30

目录

一、项目背景	- 3 -
1. 项目描述	- 3 -
2. 需求分析	- 3 -
二、技术方法及可行性	- 4 -
1. alloc_page 相关	- 4 -
1.1 参数与调用链	- 4 -
1.2 Hook 方式	- 4 -
1.3 实际测试	- 6 -
2. bpf 编程技术	- 7 -
2.1 在内核源码中编程	- 7 -
2.2 使用 BCC 工具	- 8 -
2.3 使用 bpflib & BPF CO-RE	- 8 -
3. OpenEulor 内核相关	- 9 -
三、项目规划	- 9 -
1. 项目预热阶段(6.16 - 6.30)	- 9 -
2. 项目研发第一阶段(7.01 - 8.15)	- 10 -
3. 项目研发第二阶段(8.16 - 9.30)	- 10 -
4. 未来规划与展望	- 10 -

一、项目背景

1. 项目描述

[项目描述]

内核使用 `alloc_page` 申请的内存不会显示在 `meminfo` 中，经常出现内存不够情况却不知道谁在使用，利用 `bpf` 工具统计内核 `alloc_page` 内存占用信息。

[项目产出要求]

- (1) 统计所有 `alloc_pages` 使用的内存占用信息；
- (2) 过滤出最多使用路径。

[项目技术要求]

- (1) 熟悉 `bpf`
- (2) 熟悉 linux 内核内存管理

[项目成果仓库]

✧ <https://gitee.com/openeuler/kernel>

2. 需求分析

`/proc/meminfo` 是了解 Linux 系统内存使用状况的主要接口，我们最常用的“`free`”、“`vmstat`”等命令就是通过它获取数据的，但是 Linux kernel 并没有滴水不漏地统计所有的内存分配，kernel 动态分配的内存中就有一部分没有计入 `/proc/meminfo` 中：通过 `alloc_pages` 分配的内存不会自动统计，除非调用 `alloc_pages` 的内核模块或驱动程序主动进行统计，否则我们只能看到 free memory 减少了，但从 `/proc/meminfo` 中看不出它们具体用到哪里去了。

为了实现 `alloc_pages` 函数的调用统计，我们需要借助 BPF 技术。BPF 全称是「Berkeley Packet Filter」，翻译过来是「伯克利包过滤器」。BPF 在数据包过滤上引入了两大革新：一个新的虚拟机（VM）设计，可以有效地工作在基于寄存器结构的 CPU 之上；应用程序使用缓存只复制与过滤数据包相关的数据，不会复制数据包的所有信息，最大程度地减少 BPF 处理的数据，提高处理效率。

BPF 发展到现在，名称升级为 eBPF: extended Berkeley Packet Filter。演进成一套通用执行引擎，提供了可基于系统或程序事件高效安全执行特定代码的通用能力，通用能力的使用者不再局限于内核开发者。其使用场景不再仅仅是网络分析，可以基于 eBPF 开发性能分析、系统追踪、网络优化等多种类型的工具和平台。

eBPF 由执行字节码指令、存储对象和帮助函数组成，字节码指令在内核执行前必须通过 BPF 验证器的验证，同时在启用 BPF JIT 模式的内核中，会直接将字节码指令转成内核可执行的本地指令运行，具有很高的执行效率。

目前，BPF Hooks（即BPF钩子）种类和数量都已经颇具规模，能够在内核和用户态的大部分地方加载 BPF 程序，并且可以通过 BPF Map 将分析所需要的数据传递给用户空间访问并操作，同时也为开发者们提供了一系列的 BPF Helper Function（即BPF辅助函数），对后端的内核函数进行封装，形成稳定API接口。

根据 BPF 技术提供的这些特性，我们完全可以对调用 alloc_page 的进程信息进行获取、统计与分析，实现项目需求。

二、技术方法及可行性

1. alloc_page 相关

1.1 参数与调用链

alloc_pages() 是内核中常用的分配物理内存页面的接口，用于分配一个或多个连续的物理页面，分配的页面个数只能是2的整数次幂。诸如 vmalloc、get_user_pages、以及缺页中断中分配页面，都是通过该接口分配的物理页面。而项目要求统计分析的 alloc_page() 其实是对 alloc_pages() 的封装：

```
#define alloc_page(gfp_mask) alloc_pages(gfp_mask, 0)
```

alloc_pages 调用关系如下所示：

```
alloc_pages(gfp_mask, order)
    alloc_pages_node(numa_node_id(), gfp_mask, order)
        __alloc_pages(gfp_mask, order, node_zonelist(nid, gfp_mask));
            __alloc_pages_nodemask(gfp_mask, order, zonelist, NULL);
```

alloc_pages 参数含义如下：

- (1) 内存分配掩码 gfp_mask，该标志可以分为三类：行为修饰符，区修饰符，类型修饰符。行为修饰符表示内核应当如何分配所需的内存；区修饰符表示从哪个区分配内存；类型修饰符指定所需的行为和区描述符以完成特殊类型的处理。
- (2) 分配阶数 order，指定分配页数。

通过简单了解可知，在 alloc_pages 参数内部进行分析，是无法满足我们的项目需求的，因为该函数的两个参数都没有携带关键数据。

1.2 Hook 类型

在需求分析章节中我们简要介绍了 BPF 技术，它为我们提供了一些辅助函数，能够获取进程信息。而通过 BPF 技术挂载用户代码，就需要知道如何来定位 alloc_pages。

目前，我们可以用到的 BPF 程序有两种：

Kprobe Programs: kprobes是内核提供的动态探测的功能。BPF kprobe program 类型允许我们使用BPF程序作为一个 kprobe 的执行程序，理论上 /proc/kallsyms 下面的方法都是可以 probe 程序执行的。

尽管 kprobe 可以达到跟踪的目的，但存在很多不足：1) 内核的内部 API 不稳定，如果内核版本变化导致声明修改，我们的跟踪程序就不能正常工作；2) 出于性能考虑，大部分网络相关的内层函数都是内联或者静态的，两者都不能使用 kprobe 方式探测；3) 找出调用某个函数的所有地方是相当乏味的，有时所需的字段数据不全具备；

Tracepoint Programs: 顾名思义，该类型的程序 attach 到 kernel 预先定义好的 tracepoint 上。相比于 kprobe，它是不灵活的，因为需要 kernel 预先定义好 tracepoints。但是他们是很稳定的。所有的 tracepoints 在内核的 /sys/kernel/debug/tracing/events 下面可以看到。

tracepoint 是由内核开发人员在代码中设置的静态 hook 点，具有稳定的 API 接口，不会随着内核版本的变化而变化，可以提高我们内核跟踪程序的可移植性。但是由于 tracepoint 是需要内核研发人员参数编写，因此在内核代码中的数量有限，并不是所有的内核函数中都具有类似的跟踪点，所以从灵活性上不如 kprobes 这种方式。

虽然静态 trace event 数量比较少，但是内核开发者是完全可以动手新添一个 event 的。内核可以定义自己的静态 trace event，定义代码可以放在 include/trace/events/，Trace Event 是用 include/linux/tracepoint.h 提供的宏接口来定义的。宏封装的是 tracepoint 的定义、注册和 trace 接口的定义，示例如下：

```
TRACE_EVENT(ext4_allocate_inode,
    TP_PROTO(struct inode *inode, struct inode *dir, int mode),

    TP_ARGS(inode, dir, mode),

    TP_STRUCT__entry(
        __field(    dev_t,    dev
        )
        __field(    ino_t,    ino
        )
        __field(    ino_t,    dir
        )
        __field(    __u16,    mode
        )
    ),

    TP_fast_assign(
        __entry->dev    = inode->i_sb->s_dev;
        __entry->ino     = inode->i_ino;
        __entry->dir     = dir->i_ino;
        __entry->mode    = mode;
    ),

    TP_printk("dev %d,%d ino %lu dir %lu mode 0%o",
        MAJOR(__entry->dev), MINOR(__entry->dev),
        (unsigned long) __entry->ino,
        (unsigned long) __entry->dir, __entry->mode)
    );
```

定义一个Trace Event使用了5个宏：

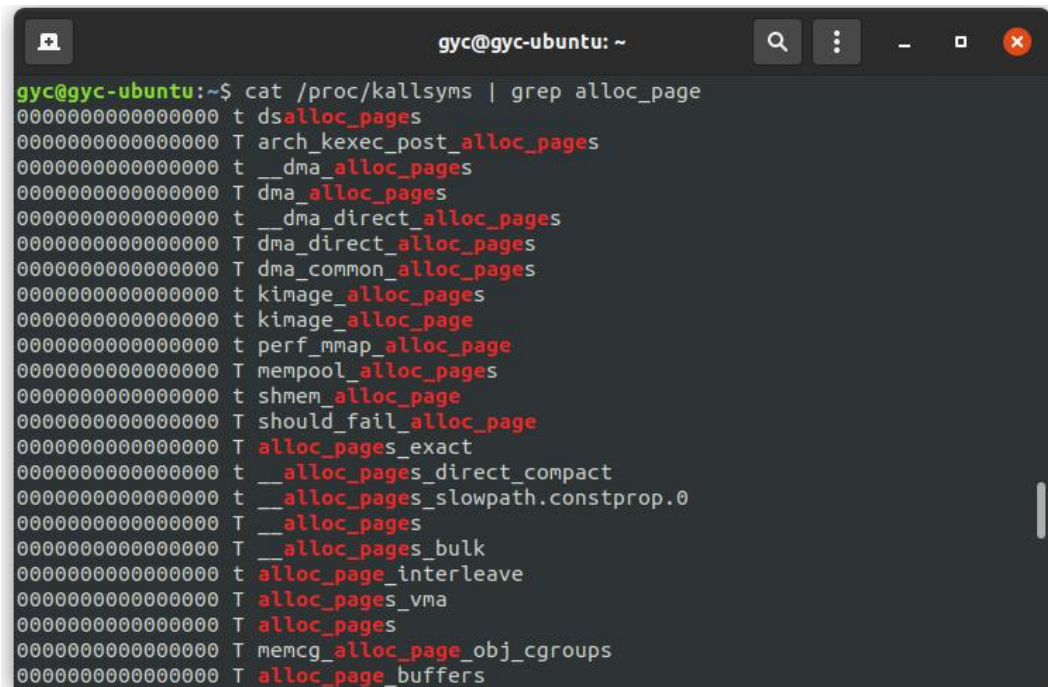
- a. TRACE_EVENT 定义一个tracepoint，它是最上层的宏，下面2-6作为它的参数
- b. TP_PROTO 定义函数原型，指定每个参数的类型
- c. TP_ARGS 列出所有参数，tracepoint的回调函数会用到
- d. TP_STRUCT__entry 定义要存入ring buffer的数据格式
- e. TP_fast_assign 对存入ring buffer的数据赋值，赋值来自前面的args
- f. TP_printk 定义打印成可读log的格式

通过在头文件中声明 trace event格式，在 c 程序中定义事件和调用 trace_xxx(args)，就可以插入一个 tracepoint，之后可以直接在内核代码中注册自己的回调函数，也可以使用 bpf 技术动态插入代码。

1.3 实际测试

在做过相关测试工作后，对于这两类追踪方式，有如下结论：

通过检索 Linux内核符号表 /proc/kallsyms 可以看到我们需要定位的符号 alloc_pages，一般在 /proc/kallsyms 并且不在 /sys/kernel/debug/kprobes/blacklist (kprobe黑名单，无法设置断点函数) 中的函数，都是可以通过 kprobe 进行动态插入探测点的：



```
gyc@gyc-ubuntu: ~  
gyc@gyc-ubuntu:~$ cat /proc/kallsyms | grep alloc_page  
0000000000000000 t dsalloc_pages  
0000000000000000 T arch_kexec_post_alloc_pages  
0000000000000000 t __dma_alloc_pages  
0000000000000000 T dma_alloc_pages  
0000000000000000 t __dma_direct_alloc_pages  
0000000000000000 T dma_direct_alloc_pages  
0000000000000000 T dma_common_alloc_pages  
0000000000000000 t kimage_alloc_pages  
0000000000000000 t kimage_alloc_page  
0000000000000000 t perf_mmap_alloc_page  
0000000000000000 T mempool_alloc_pages  
0000000000000000 t shmem_alloc_page  
0000000000000000 T should_fail_alloc_page  
0000000000000000 T alloc_pages_exact  
0000000000000000 t __alloc_pages_direct_compact  
0000000000000000 t __alloc_pages_slowpath.constprop.0  
0000000000000000 T __alloc_pages  
0000000000000000 T __alloc_pages_bulk  
0000000000000000 t alloc_page_interleave  
0000000000000000 T alloc_pages_vma  
0000000000000000 T alloc_pages  
0000000000000000 T memcg_alloc_page_obj_cgroups  
0000000000000000 T alloc_page_buffers
```

查看系统中的 tracpoint event 可以通过 perf list 或者检索 /sys/kernel/debug/tracing/events/ 路径下的内容，通过查询，并没有发现 alloc_pages 函数，但是有一个相关函数却被注册了。查看源码可知，alloc_pages 最终调用的是 _alloc_pages_nodemask，而该函数实现如下：

```

__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
                      struct zonelist *zonelist, nodemask_t *nodemask)
{
    .....
    trace_mm_page_alloc(page, order, gfp_mask, migratetype);
    .....
    return page;
}

```

我们在上面已经了解过 tracepoint 相关内容，这里正是定义了一个 hook：mm_page_alloc，正好我在events中也发现了他。不过鉴于这个hook是在 alloc_pages 调用链的末端注册的，不一定每次都会被触发（该探测点从调用链逻辑上来看是指 page 分配成功的情况），因此考虑优先级比较低，优先采用其他追踪点。

因此总的来说，如果我们想要追踪 alloc_pages 的调用，要么使用 kprobe，要么就修改内核添加新的 tracepoint 钩子，最后再考虑使用 mm_page_alloc 的 tracepoint。

2. bpf 编程技术

2.1 在内核源码中编程

LLVM Clang 编译器包含了对 eBPF 后端的支持，可以将C语言写的程序通过 LLVM Clang 编译器编译成字节码。然后可以使用bpf()系统调用函数和 BPF_PROG_LOAD命令，直接加载包含这个字节码的对象文件。为了更容易地编写 eBPF 程序，内核提供了 libbpf 库，其中包括用于加载程序、创建和操作 eBPF 对象的帮助函数（主要包含在 bpf.h 和 bpf_helper.h）。目前该编程方式的流程如下：

- (1) 读取eBPF字节码到用户应用程序中的缓冲区，并将其传递给 bpf_load_program()函数；
- (2) eBPF 程序在内核运行时，将调用 bpf_map_lookup_elem() 函数来查找 map 中的元素，并存储新值给这个元素。
- (3) 用户应用程序调用 bpf_map_lookup_elem() 函数来读取 eBPF 程序存储在内核中的值。

这种实现方式的缺点就是需要从内核源代码树中编译我们的 eBPF 程序，内核源码里包含了大量的 BPF 示例代码，就在「源码根目录/samples/bpf」文件夹下。通过编写以 kern 结尾的内核程序以及以 user 结尾的用户程序，再修改 samples/bpf/Makefile 中的内容，就可以直接得到可执行文件。其中，修改 Makefile 文件的主要作用是：

- 1) 为运行在内核空间的示例源代码（文件名称后缀为 kern.c）编译生成.o后缀的目标文件，以便加载到对应BPF提供的hook中去；

2) 为运行在用户空间的示例源代码（文件文件后缀为 user.c）编译生成可以在本机直接运行的可执行文件，以使用户可以直接运行测试。

2.2 使用 BCC 工具

BCC 全称 BPF Compiler Collection（BPF 编译器集合），该项目包括用于编写、编译和加载 eBPF 程序的工具链，以及用于调试和诊断性能问题的示例程序和久经考验的工具。在 BCC 中可以使用 Python 和 Lua 语言的作为入口进行编程。使用这些高级语言，可以编写短小但富有表现力的程序，同时具备 C 语言所缺少的全部数据操控的优势。BCC 调用 LLVM Clang 编译器，这个编译器具有 BPF 后端，可以将 C 代码转换成 eBPF 字节码。然后，BCC 负责使用 bpf() 系统调用函数，将 eBPF 字节码加载到内核中。

使用 BCC 解决移植性问题的方法如下：

- 1) 开发：将 BPF C 源码以**文本字符串**形式，嵌入（Python 编写的）用户空间控制应用（control application）；
- 2) 部署：将控制应用以源码的形式拷贝到目标机器；
- 3) 执行：在目标机器上，BCC 调用它内置的 Clang/LLVM，然后 include 本地内核头文件（需要确保本机已经安装了正确版本的 kernel-devel 包）然后现场执行编译、加载、运行。

可以发现，BCC 严重依赖本地库以及编译器，这种方式能确保 BPF 程序期望的内存布局与目标机器内核的内存布局是完全一致的，确实比第一种方式便捷很多，尤其是用于快速原型、实验和开发小工具；但当用于广泛部署生产 BPF 应用时，它存在非常明显的不足。

2.3 使用 bpflib & BPF CO-RE

使用 bcc 编写的 eBPF 代码运行时编译，不仅需要 kernel header，而且需携带 llvm/clang 相关的二进制。此外，因 kernel struct 的变更导致 memory layout 产生了变化，无法令编译生成的 eBPF 二进制运行在任意版本 Linux kernel 中，这就无法将 eBPF 二进制与用户态控制程序打包成二进制进行分发。

借助 Linux kernel 提供的 BTF、bpftool 与 libbpf，能够将 eBPF 二进制与用户态控制程序封装至单个 ELF 中，实现 CO-RE（Compile once, run everywhere），不必再担心因 memory layout 的变更导致 eBPF 二进制不再可用。只要 kernel 的 eBPF 功能具备相应的特性，它就能正常地运行在该 kernel 之上。并且借助 BTF，编译时不必需要 kernel header。

编写能够 CO-RE 的 eBPF 代码步骤如下（在 v5.10 中适用）：

- 1) 执行 `bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h` 导出当前运行的 kernel 定义的各类变量类型，该头文件包含了所有的内核类型：暴露了UAPI，通过kernel-devel提供的内部类型，以及其他一些更加内部的内核类型；
- 2) 在eBPF代码中声明的头文件如下：

```
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_tracing.h>
```

- 3) 使用Clang(版本10或更新版本)将BPF程序的源代码编译为.o对象文件
- 4) 使用 `bpftool gen skeleton <file> > <file>.skel.h` 将编译生成 .o 转换为c头文件；
- 5) 在用户态程序代码中 `#include "<file>.skel.h"` ；
- 6) 最后，编译用户空间代码，这样会嵌入BPF对象代码，后续就不用发布单独的文件。

这类方法具有适用范围更广、开发更便捷、目标文件更小等优点，是优先考虑采用的技术路线。

3. OpenEulor 内核相关

目前，OpenEulor 仓库提供了多个版本的内核代码，其中最新的系统版本 openEuler 22.03-LTS 是基于 5.10 内核构建，也就是 OLK-5.10 分支（应该没有采用最新的内容），之后开展项目研发时我会部署 OpenEulor 操作系统并且替换内核为最新的 5.10 分支内容，上面章节中我的调研和测试结果是基于本机上的 5.4 和 5.13 内核综合而来，因此相信对于 5.10 也具有很强的适用性。

三、项目规划

1. 项目预热阶段(6.16 - 6.30)

- ✧ 部署开发环境：openEuler 22.03-LTS + OLK-5.10
- ✧ 熟悉 openEuler 相关操作
- ✧ 熟悉内核开发流程，提前做好预热
 - 签署 CLA 协议
 - 熟悉社区代码风格，掌握检查工具的使用
 - 熟悉 Send Kernel patches 流程
- ✧ 继续与项目导师进行沟通，确定项目开发细节

2. 项目研发第一阶段(7.01 - 8.15)

- ◇ 实践多种 bpf 开发路线，测试稳定性及用户体验，以择取最优方案
- ◇ 完成用户态程序开发，能够统计 alloc_pages 调用信息，包括但不限于：
 - 调用进程信息
 - 分配页数量
 - 调用时间/频率等
- ◇ 设计展示方式，对统计到的信息进行格式化输出

3. 项目研发第二阶段(8.16 - 9.30)

- ◇ 对第一阶段的工作进行改进、优化
- ◇ 将工作提交至社区，以及相关事宜
- ◇ 规范代码风格、完善相关文档

4. 未来规划与展望

这是我第一次参与开源之夏，也是第一次参与到 OpenEulor 社区工作中，即使是在前期的调研工作中就已经学到了很多平时难以接触的知识，这对于我来说是一次难能可贵的经历，争取在导师指导和个人努力下以较高的完成度实现本次项目的研发，也希望能够持续参与社区工作，为开源社区做出一些贡献。