基于虚拟机调试内核

周鹏 000496

1. 引言

这次参加了国防认证考试后发现基于虚拟机大家调试内核的环境非常有必要，极短的时间、紧张的心情、不可预知的各种问题，需要一种极其方便的调试环境和快速启动内核的机制，就像 gdb 调试应用进程一样快速、方便。

除了现场调试，平时的内核分析工作非常有必要找到一种快速分析内核机制的工具和方法，现在发现虚拟机环境调试是对 bpf 的极好补充，总结来说至少有三大好处：

1)理解分析内核的原理、机制时将会更加快速、高效，能够提高我们调试内核的效率。

2)解决一些内核的非硬件方面的问题时可以在虚拟机环境中调试解决。

3)面对国防认证考试，国密考试一天时间内现场开发内核模块的任务时，借用 host 机的快速编译、虚拟机的快速启动、调试的环境等将会大大提高成功率。

本文详细介绍了如何基于虚拟机环境调试内核，它是提高我们内核开发的利器，主要基于几个具体的案例介绍如何调试内核，最后将我们已经搭建好的虚拟机环境共享出来，欢迎大家使用，提高效率。

2. 虚拟机环境

首先感谢服务器团队徐振海帮忙提供了 arm 版本的虚拟机调试环境，成都国防团队柏鑫和明宇帮忙提供了 x86 的调试环境。

2.1. arm 环境搭建

scp uos@10.20.52.198:~/zhoupeng/min-arm-vm.tar.gz .
passwd:1
解压后就可以使用了

下载下来后，先解压，确保安装了 qemu，然后就可以调试内核了，方法如下：
tar xvf min-arm-vm.tar.gz
cd min-arm-vm

启动内核，但以调试方式等待 gdb 连接调试
./startvm.sh

gdb 远程连接调试
./dbgme.sh

看到 Reading symbols from ark-kernel-4.19/vmlinux ... done.的打印后，
在客户端输入
target remote:1234

此时可以开始调试了，最好的确认办法就是
b start_kernel

输入 c
开始运行，此时停止到 start_kernel，可以开始调试内核了

下面详细讲解一下

2.1.1. 启动虚拟机

提供一个启动虚拟机的脚本 startvm.sh，它的内容如下：

```
#!/bin/sh
qemu-system-aarch64      \
-machine virt,virtualization=true,gic-version=3          -nographic          -m
size=1024M   \
-cpu cortex-a57          -smp 2       \
-kernel   /home/uos/zhoupeng/kernel-4.19/arm-kernel-4/arch/arm64/boot/Image
\
-initrd busybox-1_27_2/rootfs.cpio.gz     \
--append "console=ttyAMA0 rdinit=/linuxrc " \
-S -s
```

2.1.2. 命令理解

qemu-system-aarch64: arm64 版本的虚拟机，通过 file 命令可以看出来

file /usr/bin/qemu-system-aarch64

/usr/bin/qemu-system-aarch64: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (GNU/Linux), dynamically linked, interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0,
BuildID[sha1]=018a921adbc76be5b9d6e807045bbdc75821ecbe, stripped

-m size=1024: 虚拟机配置 1024M 的内存，可以尝试修改

-cpu : 配置 cpu 的名字，arm 版本发现目前只支持设置此名字
-smp 2: 可以指定模拟多少个 cpu,理论上可以 host 机器的 cpu 个数一样多

-kernel: arm-kernel-4.19/arch/arm64/boot/Image
指定内核镜像文件

file arm-kernel-4.19/arch/arm64/boot/Image
arm-kernel-4.19/arch/arm64/boot/Image: MS-DOS executable

-initrd: busybox-1_27_2/rootfs.cpio.gz

指定根文件系统

--append: "console=ttyAMA0 rdinit=/linuxrc "
指定内核启动参数

2.1.3. gdb 连接服务端

dbgme.sh

```
/usr/bin/gdb \
-ex "file arm-kernel-4.19/vmlinux" \
-q
```

-ex 指定需要调试的文件，这儿为 linux 的 elf 文件，它就像进程 elf 文件一样

```
target remote:1234
b start_kernel
c
```
启动后就可以调试内核了

```
Thread 1 hit Breakpoint 1, start_kernel () at init/main.c:535
535             set_task_stack_end_magic(&init_task);
(gdb) n
536             smp_setup_processor_id();
(gdb)
539             cgroup_init_early();
(gdb)
541             local_irq_disable();
(gdb) n
542             early_boot_irqs_disabled = true;
(gdb)
548             boot_cpu_init();
(gdb)
```

OK，到这儿为止，就说明虚拟机的环境 OK 了，可以调试了。

不过调试内核时，通常都有执行某个命令、或启动一个进程、或加载一个 ko 来触发内核相关函数运行的需求，这个可以通过将直接编译好的进程、ko 放到 busybox-1_27_2/_install 目录，然后重新打包生成新的根文件系统 rootfs.cpio.gz 就可以了，请看 3.1 节理解 MMU 中的详细讲解。

2.2. x86 环境搭建

scp uos@10.20.52.198:~/zhoupeng/min-x86-vm.tar.gz .

passwd :1
tar zxvf min-x86-vm.tar.gz
cd kvm-machine/

2.2.1. 服务端

sudo su
./dbgvm.sh
该脚本默认让内核以调试方式启动，因此需要 gdb 客户端连接上来后请求运行

2.2.2. 客户端

cd kvm-machine/share/linux-mainline
gdb ./vmlinux

target remote:1234
该命令执行后，会看到如下提示信息：
Remote debugging using :1234
0x000000000000fff0 in entry_stack_storage ()

2.2.3. 调试断点验证

x86 环境下需要设置硬件端点调试，如下：
hb start_kernel

按 c 时就会在 start_kernel 停下来，然后就可以单步跟踪执行了
Thread 1 hit Breakpoint 1, start_kernel () at init/main.c:576
576   {
(gdb) n
580   set_task_stack_end_magic(&init_task);
(gdb)
581   smp_setup_processor_id();
(gdb)
584   cgroup_init_early();

(gdb)
586 local_irq_disable();

3. arm 虚拟机调试举例

3.1. 理解 mmu

mmu 是 linux kernel 一个非常重要的概念，它的本质就是根据虚拟地址查找物理地址，目的是为了安全，同时也做到了内存真正使用时才申请。

根据虚拟地址查找物理地址的本质是查找页表，这个查找是 cpu 执行时 mmu 硬件单元去查的，查找时内核是没有参与的，但是内核完成了页表的写入，只有当内核按照页表映射的规范完成了写，cpu 才能读。否则就会报缺页异常，让内核去处理，内核要么处理异常要么抛出异常(如段错误)。

内核对于用户态虚拟地址和内核态虚拟地址的处理原则是不一样的，对于用户态地址，就是使用时才因为缺页才去完成物理内存的申请、同时完成页表的映射；对于内核态虚拟地址，在申请出虚拟地址时顺便就完成了页表的映射。

本节基于 vmalloc 中的调用场景介绍了页表映射的本质，页表代码很多，但本质一致，为了描述清楚问题，只介绍了最后一层 pte 页表的写入。

3.1.1. 构建 ko

最好的方式是自己写一个调用 vmalloc 的调用，看看它是如何在申请完物理内存后完成虚拟地址到物理地址的映射，然后将虚拟地址返回给调用者使用的。

可以自己写一个 ko，然后在 ko 中调用 vmalloc，ko 模块的编写请参考 drivers/uos/hellodrv.c，实现的主要函数如下：

static int hello_open(struct inode *inode,struct file *file)

static int hello_release(struct inode *inode,struct file *file)

static ssize_t hello_read(struct file *filp,char __user *buf,size_t count,loff_t *offp)

static ssize_t hello_write(struct file *flip,const char __user *buf,size_t count,loff_t *offp)

static int __init hello_init(void)

static void __exit hello_exit(void)

drivers/uos/Makefile 中增加如下编译代码：

obj-m += hellodrv.o

make modules SUBDIRS=drivers/uos

最后会生成 hellodrv.ko

3.1.2. 安装 ko

执行 arm-kernel-4.19/pkgko.sh

已知该脚本的内容如下:
cp drivers/uos/hellodrv.ko ../busybox-1_27_2/_install/ko/
cd ../busybox-1_27_2
./build-pkg
cd -

build-pkg 的内容如下：
cd _install
find . | cpio -o -H newc |gzip > ../rootfs.cpio.gz

它的功能就是将刚刚编译出来的 ko copy 到 busybox 制作的根文件系统中，即 copy 到_install 目录下面，然后将_install 目录下面的所有文件打包。

3.1.3. 调试 ko

虚拟机方式运行内核
./startvm.sh

gdb 连接调试内核
./gdbme.sh

run 命令启动

此时虚拟机方式启动的内核看到如下打印
[0.008371] [manager_init] start manager_init
[0.009013] UOS Manager initialized: uosmanager
[0.215070] register_cpu_capacity_sysctl: too early to get CPU1 device!

Please press Enter to activate this console.

控制台中加载 ko
insmod /ko/hellodrv.ko

创建设备文件

mknod /dev/myhello c 239 0

此时将会看到/dev/myhello 文件被创建

cat /dev/myhello 就会触发 hello_open 函数的调用了，由于 hello_open 函数中增加了 vmalloc 函数的调用，因此也就可以调试 vmalloc 函数了

3.1.4. 调试 vmalloc 函数

b vmalloc

控制台中 cat /dev/myhello

享受一下 gdb 带来的调试快乐:

当 vmalloc 函数被停止下来后，看到如下调用堆栈:

```
#0  vmalloc (size=196608) at ./arch/arm64/include/asm/jump_label.h:34
#1  0xffff000000c20010 in ?? ()
#2  0xffff00000829db38 in do_dentry_open (f=0xffff000000c30058, inode=0xffff000000c40390,
      open=0xffff0000082a6e98 <chrdev_open>) at fs/open.c:796
#3  0xffff00000829f010 in vfs_open (path=<optimized out>, file=<optimized out>)
      at ./include/linux/dcache.h:545
#4  0xffff0000082b2864 in do_last (nd=0xffff800020993d28, file=0xffff80002063e400,
      op=0xffff800020a74268)  at fs/namei.c:3421
#5  0xffff0000082b2ee8 in path_openat (nd=0xffff800020993d28, op=0xffff800020993e4c, flags=65)
       at fs/namei.c:3536
#6  0xffff0000082b41f8 in do_filp_open (dfd=<optimized out>, pathname=<optimized out>,
      op=0xffff000000c20000) at fs/namei.c:3567
#7  0xffff00000829f2f4 in do_sys_open (dfd=546913868, filename=<optimized out>, flags=<optimized out>,
      mode=<optimized out>) at fs/open.c:1088
#8  0xffff00000829f3d8 in __do_sys_openat (mode=<optimized out>, flags=<optimized out>,
      filename=<optimized out>, dfd=<optimized out>) at fs/open.c:1115
#9  __se_sys_openat (mode=<optimized out>, flags=<optimized out>, filename=<optimized out>,
      dfd=<optimized out>)  at fs/open.c:1109
#10 __arm64_sys_openat (regs=<optimized out>) at fs/open.c:1109
#11 0xffff000008095218 in __invoke_syscall (syscall_fn=<optimized out>, regs=<optimized out>)
     at arch/arm64/kernel/syscall.c:48
#12 invoke_syscall (syscall_table=<optimized out>, sc_nr=<optimized out>, scno=<optimized out>,
      regs=<optimized out>) at arch/arm64/kernel/syscall.c:48
#13 el0_svc_common (regs=0x3, scno=<optimized out>, sc_nr=<optimized out>, syscall_table=0x1)
      at arch/arm64/kernel/syscall.c:114
#14 0xffff000008095384 in el0_svc_handler (regs=<optimized out>) at arch/arm64/kernel/syscall.c:174
#15 0xffff000008083f88 in el0_svc () at arch/arm64/kernel/entry.S:904
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

连续几次 s 命令进入__vmalloc_node_range 函数,

(gdb) s

__vmalloc_node_range (size=196608, align=1, start=18446462598867058688, end=18446598800735666176, gfp_mask=6291648,

prot=..., vm_flags=0, node=-1, caller=0xffff000000c20010) at mm/vmalloc.c:1742

1742 if (!size || (size >> PAGE_SHIFT) > totalram_pages)

从__vmalloc_node_range 出发到完成物理地址到虚拟地址映射的调用路径如下:

__vmalloc_node_range -->

__vmalloc_area_node --> map_vm_area --> vmap_page_range --> vmap_page_range_noflush

vmap_page_range_noflush 实现了地址映射

该函数代码如下:

```
static int vmap_page_range_noflush(unsigned long start, unsigned long end,
                                   pgprot_t prot, struct page **pages)
{
    pgd_t *pgd;
    unsigned long next;
    unsigned long addr = start;
    int err = 0;
    int nr = 0;

    BUG_ON(addr >= end);
    pgd = pgd_offset_k(addr);//从 pgd 出来
    do {
        next = pgd_addr_end(addr, end);
        //最终会将物理地址写到 pte 表中
        err = vmap_p4d_range(pgd, addr, next, prot, pages, &nr);
        if (err)
            return err;
    } while (pgd++, addr = next, addr != end);

    return nr;
}
```

3.1.5. pgd 地址的获取

已知:

```
pgd = pgd_offset_k(addr)
```

pgd_offset_k 的定义如下(arch/arm64/include/asm/pgtable.h):

```
#define pgd_offset_k(addr)   pgd_offset(&init_mm, addr)

#define pgd_offset(mm, addr)      (pgd_offset_raw((mm)->pgd, (addr)))

#define pgd_offset_raw(pgd, addr)    ((pgd) + pgd_index(addr))

#define pgd_index(addr)       (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))
```

a)计算 PTRS_PER_PGD

```
#define PTRS_PER_PGD          (1 << (VA_BITS - PGDIR_SHIFT))

#define    PGDIR_SHIFT                  ARM64_HW_PGTABLE_LEVEL_SHIFT(4    -
```

CONFIG_PGTABLE_LEVELS)
CONFIG_PGTABLE_LEVELS=3

#define ARM64_HW_PGTABLE_LEVEL_SHIFT(n) ((PAGE_SHIFT - 3) * (4 - (n)) + 3)

因为#define PAGE_SHIFT CONFIG_ARM64_PAGE_SHIFT
CONFIG_ARM64_PAGE_SHIFT=16

所以
ARM64_HW_PGTABLE_LEVEL_SHIFT(4-3)=ARM64_HW_PGTABLE_LEVEL_SHIFT(1)
=(16-3)*(4-1)+3=42

VA_BITS = CONFIG_ARM64_VA_BITS=48
所以 PTRS_PER_PGD = 1<< (48-42)=1<<6=64
所以每个 PGD 地址基于虚拟地址的高 6 位的值为索引到 pgd 表中取得

b)计算 PGDIR_SHIFT
PGDIR_SHIFT=ARM64_HW_PGTABLE_LEVEL_SHIFT(4-3)=ARM64_HW_PGTABLE_
LEVEL_SHIFT(1)=(16-3)*(4-1)+3=42

c)计算 pgd_index(addr)
#define pgd_index(addr) (((addr) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))

pgd_index(addr) = (addr >> 42) & (63)

(gdb) p /x addr
$1 = 0x6000c0

所以 pgd_index(0x6000c0) = 0

d)计算 pgd_offset_k(mm,addr)
#define pgd_offset_k(addr) pgd_offset(&init_mm, addr)

#define pgd_offset(mm, addr) (pgd_offset_raw((mm)->pgd, (addr)))

#define pgd_offset_raw(pgd, addr) ((pgd) + pgd_index(addr))

((&init_mm)->pgd) + 0

e)基于预编译展开
pgd = ((((&init_mm)->pgd) + ((((addr)) >> ((16 - 3) * (4 - (4 - 3)) + 3)) & ((1
<< ((48) - ((16 - 3) * (4 - (4 - 3)) + 3))) - 1))));

pgd = ((((&init_mm)->pgd) + ((((addr)) >> ((16 - 3) * (4 - (4 - 3)) + 3)) & 63)));
= ((((&init_mm)->pgd) + ((((addr)) >> 42) & 63)));
=((((&init_mm)->pgd) + ((addr >> 42) & 63)));

=((((&init_mm)->pgd) + 0));
=((((&init_mm)->pgd) + 0));

f)最终结果确认
(gdb) p (&init_mm)->pgd
$15 = (pgd_t *) 0xffff000009b80000
(gdb) p pgd
$16 = (pgd_t *) 0xffff000009b80000

所以 pgd 取的是 init_mm->pgd 的第一个 pgd，这个 pgd 是根据虚拟地址 addr 取出来的，对于 48 位的虚拟地址，43~48 位就是 pgd 索引。

3.1.6. pte 表项的写入

vmap_p4d_range -->vmap_pud_range --> vmap_pmd_range --> vmap_pte_range

```
static int vmap_pte_range(pmd_t *pmd, unsigned long addr,
        unsigned long end, pgprot_t prot, struct page **pages, int *nr)
{
    pte_t *pte;

    /*
     * nr is a running index into the array which helps higher level
     * callers keep track of where we're up to.
     */

    pte = pte_alloc_kernel(pmd, addr);//pte 表中保存了物理地址，pte 表本身需要申请内存
    if (!pte)
        return -ENOMEM;
    do {
        struct page *page = pages[*nr];

        if (WARN_ON(!pte_none(*pte)))
            return -EBUSY;
        if (WARN_ON(!page))
            return -ENOMEM;
```

```
            set_pte_at(&init_mm, addr, pte, mk_pte(page, prot));//
            (*nr)++;
        } while (pte++, addr += PAGE_SIZE, addr != end);
        return 0;
}
```

3.1.7. mk_pte 的理解

mk_pte 的定义如下:

```
#define mk_pte(page,prot)    pfn_pte(page_to_pfn(page),prot)

#define __page_to_pfn(page) (unsigned long)((page) - vmemmap)

#define   vmemmap                        ((struct   page   *)VMEMMAP_START   -
(memstart_addr >> PAGE_SHIFT))

#define pfn_pte(pfn,prot)    \
      __pte(__phys_to_pte_val((phys_addr_t)(pfn)      <<      PAGE_SHIFT)      |
pgprot_val(prot))

typedef u64 phys_addr_t
```

a)根据 page 得到 pfn

```
page_to_pfn(page)
```

b)根据 pfn 得到物理地址

```
(phys_addr_t)(pfn) << PAGE_SHIFT
```

c)将物理地址转换成 pte 值

当物理地址为 48 位时的定义如下:

```
 #define __phys_to_pte_val(phys) (phys)
```

直接将物理地址当作 pte 值保存起来了

d)_pte 处理 pte 值

```
typedef struct { pteval_t pte; } pte_t;
typedef u64 pteval_t;
 #define __pte(x)      ((pte_t) { (x) } )
```

pte_t 是一个数据结构, pte 本身是一个 64 位的整数值，因此将它保存到了数据结构之中。

所以 mk_pte 就是根据 page 地址得到物理地址，然后将物理地址保存到 pte_t 这个数据结构中，mk_pte 返回的就是 pte_t 这个数据结构

set_pte_at(
&init_mm, addr, pte,
((pte_t) { (((phys_addr_t)((unsigned long)((page) - ((struct page *)
((((((0xffffffffffffffffUL))) - ((((1UL))) << ((48) - 1)) + 1) - ((((1UL))) << ((48) - 16 - 1 + 6)))
- (memstart_addr >> 16)))) << 16) | ((prot).pgprot)) })
);
=set_pte_at(
&init_mm, addr, pte,
((pte_t) { (((phys_addr_t)
((unsigned long)((page)-(struct page *)0xffff7fdffff00000)) << 16) |
((prot).pgprot)) })
);

假设 page=0xffff7fe000084b40
那么 pfn = (struct page*)0xffff7fe000084b40 - (struct page *)0xffff7fdffff00000
=0x612d
phyaddr=0x612d << 16 = 0x612d0000

因为 prot=0x68000000000713 phyaddr= 0x612d0000
所以最终 mk_pte 返回的 pte 的值为: 0x68000000000713 | 0x612d0000
= 0x680000612d0713

总结：
这儿物理内存的开始地址为 memstart_addr=0x4000 0000，所以开始的物理页编号为 0x4000

计算出第 0 页的 struct page 的虚拟地址
((struct page *)0xffff7fe000000000 - (memstart_addr >> 16))
= (page *) 0xffff7fdffff00000

所以当一页的 struct page=0xffff7fe000084b40，可以根据第 0 页的 struct page 地址得知是第几页的，即得到了物理页编号.
(0xffff7fe000084b40 - 0xffff7fdffff00000)>>6=0x612d

所以根据 struct page 地址计算出物理地址的原理就是一个相对比例的计算:struct page 地址之间的偏移就是物理页编号的偏移；相对于第 0 页物理地址的偏移计算出来的偏移就是该页的物理页编号 pfn 了，所以这儿的 0x612d 就是物理页编号了。

3.1.8. set_pte_at

set_pte_at 最终调用的是下面这各语句：
set_pte(ptep, pte);

pte 是 mk_pte 返回的值 0x680000612d0713
ptep 是 pte 页表地址
所以最终将 pte_t 类型的值保存到了 pte 页表中，这就是页表映射的本质

3.1.9. gdb 调试方式理解 pte 页表的写入

上面都是基于理解宏代码来计算的，比较麻烦，在不太熟悉的情况下也不知道计算是否正确.

纯调试环境下，如果每个变量的值都能在 gdb 中查看到，那么调试是非常方便的，但由于内核编译时进行了优化，很多变量都被优化了，它们的值都被放到了寄存器中，无法准确得到寄存器与变量的对应关系。

下面介绍一种基于 gdb 调试的方法，先想办法得到其地址，然后基于 gdb 查看地址的方式来调试，让理解变得更加快速，计算变得更加简单。

以调试 vmap_pte_range 为例，再该函数中增加如下代码：

```
int g_debug_mmu = 0;
141 static int vmap_pte_range(pmd_t *pmd, unsigned long addr,
142             unsigned long end, pgprot_t prot, struct page **pages, int *nr)
143 {
144         pte_t *pte;
145
151         pte = pte_alloc_kernel(pmd, addr);
152         if (!pte)
153             return -ENOMEM;
154         do {
155             struct page *page = pages[*nr];
156
157             if (WARN_ON(!pte_none(*pte)))
158                 return -EBUSY;
159             if (WARN_ON(!page))
160                 return -ENOMEM;
161             if (g_debug_mmu & 1) {//新增的调试代码
162                     int len = snprintf(g_debug_acBuf, sizeof(g_debug_acBuf), "addr=0x%lx pte=0x%lx, page=0x%lx pages=0x%lx nr=0x%lx, prot=0x%lx pmd=0x%lx page_to_pfn=0x%lx mk_pte=0x%lx", addr, pte, page, pages, nr, prot, pmd, page_to_pfn(page), mk_pte(page, prot));
163                 printk("len=%d acBuf=%s\n", g_debug_acBuf);
164             }
165             set_pte_at(&init_mm, addr, pte, mk_pte(page, prot));
166
167             (*nr)++;
168         } while (pte++, addr += PAGE_SIZE, addr != end);
169         return 0;
170 }
```

其中 g_debug_acBuf 是一个全局变量
char g_debug_acBuf[256];

drivers/uos/hellodrv.c 中增加如下代码:

```
static int hello_open(struct inode *inode,struct file *file)
{
        char*   psz = (char*)vmalloc(64*1024*3);
        strcpy(psz, "111111111111111112222222");
        printk("myhello open, psz=%s\n", psz);
        vfree(psz);
        return 0;
}
```

当 cat /dev/myhello 文件时就会调用 hello_open 函数，这个函数中调用 vmalloc 只是为了我们方便调试 mmu.

安装 ko 并触发 vmalloc 的调用
insmod /ko/hellodrv.ko
mknod /dev/myhello c 239 0

b vmap_pte_range
或者
b mm/vmalloc.c:151

在控制台中执行:
cat /dev/myhello

此时 gdb 会在 mm/vmallo.c:151 停下来

当运行到 161 行时
p g_debug_mmu=1
当执行了 162 行的语句后
(gdb) p g_debug_acBuf
$3 = "addr=0xffff00000c1a0000 pte=0xffff80003ffd60d0, page=0xffff7fe000084080 pages=0xffff800020c56280 nr=0xffff800020987aac, prot=0x68000000000713 pmd=0xffff80003ffe0000 page_to_pfn=0x6102 mk_pte=0x6800006"...
(gdb) p g_debug_acBuf[150]@100
$4 = "ff80003ffe0000 page_to_pfn=0x6102 mk_pte=0x68000061020713", '\000' <repeats 42 times>
(gdb) p *(long*)0xffff80003ffd60d0
$5 = 0

mk_pte 返回的值就是要写到 pte 页表中的值， pte 页表的地址，mk_pte 返回的值都用红色标注了。

由于 165 行的代码还没有执行，因此 pte 页表中的值为 0

执行 set_pte_at 函数

```
(gdb) n
165                             set_pte_at(&init_mm, addr, pte, mk_pte(page,
prot));
(gdb)
167                         (*nr)++;
```

这个是执行后的结果，可以看到 mk_pte 计算出来的值已经被写到了 pte 页表

```
(gdb) p /x *(long*)0xffff80003ffd60d0
$7 = 0x68000061020713
```

3.1.10.mk_pte 理解

mk_pte 到底返回的是什么意义的值呢?

从上面的分析可以看到它就是 prot 和 phy_addr 相或后的值

这儿 prot=0x68000000000713，表示页的属性

page_to_pfn=0x6102，所以 phy_addr=0x6102 <<16=0x61020000

prot|phy_addr=0x68000000000713|0x61020000 = 0x68000006102713

a)最终写入 pte 页表中的值是页的属性和页的物理地址，这个是 mmu 的本质，内核写入，cpu 读取。如果 pte 页表中没有物理地址或者页属性，则 mmu 会抛出缺页异常，让内核来处理

b)由于内核的很多符号都优化了，虽然保存到了寄存器中，但是看出保存到了哪个寄存器中，为了查看变量的值，可以将这些变量的地址 snprintf 到一个缓冲区中，然后在 gdb 中显示该缓存区的值。

3.2. 理解 Page 的属性与释放

3.2.1. 构建自己的调试函数

学习过 c 库内存管理就会知道，free 释放内存时通常都不会回收内核，而是回收到自己的内存池中，只有当调用 brk 或者 munmap 才会回收内核。

内核的内存管理也非常庞大，经常听说 kmalloc、vmalloc、直接 memblock 映射内存，也经常听说 slab 内存管理，__alloc_pages_nodemask 申请页内存、也听说过 buddy 系统，内存 zone 的概念，如果延伸到服务器还要涉及 numa 架构。

实际上 kmalloc、vmalloc 只是对外接口，slab 只是实现 kmalloc 的内部内存管理，内核真正的内存管理是页管理，就像 c 看的内存管理内存块管理基于 8、16、32、64、...等大小做等块内存管理 一样，内核基于 zone 的采用不同页大小的(order=0、1、2、3...)来分配和合并，这个就是著名的内存伙伴系统。

内存伙伴系统依赖的最核心的机制就是内存是也页(Page)来管理的，Page 不仅决定内核的内存是如何管理的，还决定了虚拟地址和物理地址之间的转换算法，简单来说在内核内部虚拟地址、物理地址、Page 地址是可以简单地线性相互转换的。

由于 Page 决定内存管理的本质，因此本节稍微介绍一下 Page 的属性和释放。具体思路就是自己写几个调试函数，基于虚拟地址得到 Page 地址、基于 Page 的属性快速判断该页的状态。

```
unsigned long zpLmVirt(unsigned long virt) {
    unsigned long reloc = RELOC_HIDE(virt, 0);
    unsigned long phys = __virt_to_phys(reloc);
    unsigned long newVirt = __phys_to_virt(phys);
    char acBuf[128];//一种调试技巧方便调试看中间值
    snprintf(acBuf, sizeof(acBuf), "[reloc=0x%lx phys=0x%lx
newVirt=0x%lx]end", reloc, phys, newVirt);
    printk("%s\n", acBuf);
    return newVirt;
}

struct page* zpVirt2Page(void *addr) {
    void *pos = (void *)PAGE_ALIGN((unsigned long)addr);
    struct page* mypage = virt_to_page(pos);
    return mypage;
}
```

zpLmVirt：根据 gdb 中看到虚拟地址或者是 kallsyms 中看到的虚拟地址转换成内核内部真正用到的虚拟地址。

zpVirt2Page:跟踪转换出来的虚拟地址得到 Page 地址，这样就可以查看此页的属性了

3.2.2. gdb 调试页属性

假设内核已经完全启动了

```
(gdb) p __init_begin
Cannot access memory at address 0xffff000009200000
```

从这个打印可以看到，__init_begin 已经无法访问了

```
(gdb) call zpLmVirt (0xffff000009200000)
$1 = 18446603336240070656
(gdb) p /x $1
```

$2 = 0xffff800001200000
$2 表示真正的虚拟地址

(gdb) call zpVirt2Page(0xffff800001200000)
$3 = (page *) 0xffff7fe000004800
得到了页地址

可以查看页属性了
(gdb) p /x ((struct page*)$3)->flags
$4 = 0xffff0000000000
(gdb) p /x ((struct page*)$3)->page_type
$5 = 0xffffff7f
(gdb)
下面来理解一下页属性，特别是当此页还没有释放时页的属性是咋样的对比确认一下

3.2.3. gdb 调试释放之前的页属性

在内核释放内存页之前查看页属性，如下操作

Thread 1 hit Breakpoint 2, free_initmem () at arch/arm64/mm/init.c:628
628 free_reserved_area(lm_alias(__init_begin),

当运行到 free_initmem 后查看__init_begin 的地址，此时它还是可以查看的
(gdb) p __init_begin
$4 = 0xffff000009200000 <stext> "\b"

转换成内核内部的虚拟地址
(gdb) call zpLmVirt($4)
$5 = 18446603336240070656

根据虚拟地址获取 Page 地址
(gdb) call zpVirt2Page($5)
$6 = (struct page *) 0xffff7fe000004800

查看页属性
(gdb) p /x ((struct page *) 0xffff7fe000004800)->flags
$7 = 0xffff0000000800
(gdb) p /x ((struct page *) 0xffff7fe000004800)->page_type
$8 = 0xffffffff

通过对比可以发现页的属性发生了变化，首先是 flags 属性发生变化了，上面红色字体标志的地方:

”8”表示 PG_reserved，即这个内存已经被内核使用了，释放后没有了，变成了 0。

“f”变成了 7，因为去掉了 0x80 属性，它表示 PG_buddy，释放后删除了此标志，申请时待了此标志，所有当一个页被申请出来时，带上了 0x80 标志，这个标识是决定页内存是已经被申请还是被释放的重要信息。

有 0x80:已经被申请出来了；

没有 0x80:已经被释放了

3.2.4. 页属性设置代码确认

从 free_initmem 出发，最后发现会调用__free_reserved_page 函数，红色代码就是清除了页的 PG_reserved 标志

```
static inline void __free_reserved_page(struct page *page)
{
    ClearPageReserved(page);
    init_page_count(page);
    __free_page(page);
}
```

进一步往下跟踪，当最终调用到__free_one_page 时，调用了 set_page_order 函数，该函数表面上是调用的函数是 Set PageBuddy 属性，但从实现的函数来看它是将这个标志去除了，即将 0x80 去掉了，所以当页被释放后，页属性是没有 PG_buddy=0x80 属性的。

```
static inline void set_page_order(struct page *page, unsigned int order)
{
    set_page_private(page, order);
    __SetPageBuddy(page);
}
```

3.3. 根据 current 确定当前运行进程

内核中在任何地方都可以当前运行的进程是谁，以 arm 版本为例，这是因为当前进程的 struct task_struct 地址被放到了 cpu 的 SP_EL0 寄存器中。

看一下 arm 版本的实现就可以知道了，如下：

```
#define current get_current()

static __always_inline struct task_struct *get_current(void)
{
    unsigned long sp_el0;

    asm ("mrs %0, sp_el0" : "=r" (sp_el0));
```

```
    return (struct task_struct *)sp_el0;
}
```

从 SP_EL0 寄存器读取了进程的 task_struct 地址。
举例
虚拟机启动起来后，加载 ko，触发 vmalloc 函数的调用

```
 # insmod /ko/hellodrv.ko
 # mknod /dev/myhello c 239 0
```

对 vmalloc 设置断点：

```
(gdb) b vmalloc
Breakpoint 1 at
 0xffff00000825ef70: file ./arch/arm64/include/asm/jump_label.h, line 34.
```

打开/dev/myhello 文件，触发 vmalloc 函数的调用

```
 # cat /dev/myhello
```

gdb 环境查看当前运行进程

```
(gdb) p ((struct task_struct*)$SP_EL0)->comm
$1 = "cat\000xrc\000\060\000\000\000\000\000\000"
```

可以看到是 cat 进程触发了 vmalloc 的调用

4. x86 虚拟机调试

4.1. 调试调度器

为了学习调度器，自己移植裁剪了一个 RT 调度器，需要让应用用自己的调度器来运行，结果内核启动就死机了，本节描述了如何基于 gdb 调试快速找到死机的点

4.1.1. 调度器的实现

移植实现的调度器在 kernel/sched/myp.c， 称之为自己的抢占式调度器

a)首先让 deadline 调度器的下一个调度指向自己的调度器 myp_sched_class:

```
const struct sched_class dl_sched_class = {
        .next                   = &myp_sched_class,
```

b)其次让 myp 调度器指向了 FAIR 调度器

```
const struct sched_class myp_sched_class = {
        .next                   = &fair_sched_class,
```

c)编译调度器

kerenel/sched/Makefile 中也增加了 myp.o 的编译

```
obj-y += idle.o fair.o rt.o deadline.o   myp.o
```

d)内核中所有使用 rt 调度器的地方改成使用 myp 调度器

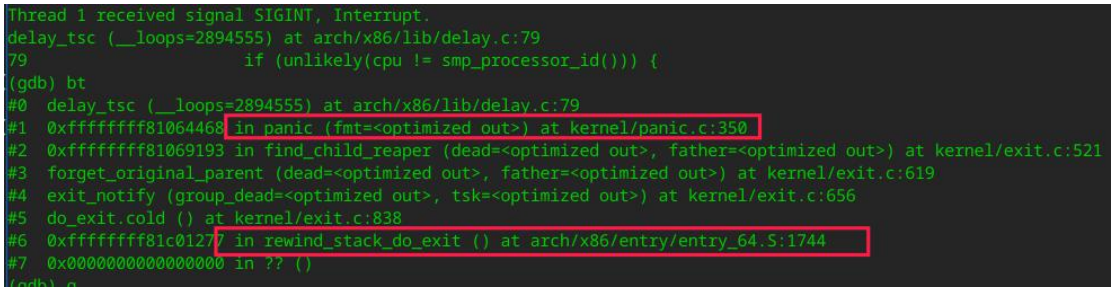
```
        else if (rt_prio(p->prio))
-               p->sched_class = &rt_sched_class;
+                p->sched_class = &myp_sched_class;
```

详细信息请用下面的命令查看

```
git show 14b136ee39ab8e853379db19ba46d929552f4e7a
```

4.1.2. 调度器的运行

完成上面的代码编写后，按照上面的步骤启动虚拟机和调试器，结果内核死机了，死机的堆栈如下：



看到上面的堆栈只能知道内核恐慌了，从 do_exit 出发，那么这背后的逻辑是什么呢？从哪儿开始分析呢？

在不知道背后的逻辑的情况下，最快的办法就是对该堆栈中能看到的最底层的函数 rewind_stack_do_exit 出发，寻找堆栈的调用情况。

4.1.3. 跟踪 rewind_stack_do_exit

退出虚拟机重新运行，结果如下：

```
Thread 1 hit Breakpoint 2, rewind_stack_do_exit () at arch/x86/entry/entry_64.S:1738
1738            xorl    %ebp, %ebp
(gdb) bt
#0  rewind_stack_do_exit () at arch/x86/entry/entry_64.S:1738
#1  0xffffffff8101f530 in oops_end (flags=<optimized out>, regs=<optimized out>, signr=<optimized out>)
    at arch/x86/kernel/dumpstack.c:364
#2  oops_end (flags=70, regs=0xffffc90000013d08, signr=11) at arch/x86/kernel/dumpstack.c:331
#3  0xffffffff8101c34c in do_trap_no_signal (error_code=<optimized out>, regs=<optimized out>,
    str=<optimized out>, trapnr=<optimized out>, tsk=<optimized out>) at arch/x86/kernel/traps.c:212
#4  do_trap (trapnr=<optimized out>, signr=4, str=<optimized out>, regs=0xffffc90000013d08, error_code=0,
    sicode=2, addr=0xffffffff810a54a1 <rq_offline_myp+513>) at arch/x86/kernel/traps.c:251
#5  0xffffffff8101c6ab in do_error_trap (regs=<optimized out>, error_code=<optimized out>, str=<optimized out>,
    trapnr=<optimized out>, signr=<optimized out>, sicode=<optimized out>,
    addr=0xffffffff810a54a1 <rq_offline_myp+513>) at arch/x86/kernel/traps.c:278
#6  0xffffffff8101ca51 in do_invalid_op (regs=0xffffc90000013d08, error_code=0) at arch/x86/kernel/traps.c:291
#7  0xffffffff81c00bbe in invalid_op () at arch/x86/entry/entry_64.S:1028
#8  0xffff8881f9be7bc0 in ?? ()
#9  0xffffffffc7602680 in ?? ()
Backtrace stopped: Cannot access memory at address 0x27540
```

基于上面的原理继续跟踪调用 invalid_op 的地方

4.1.4. 跟踪 invalid_op 的调用

```
Thread 1 hit Breakpoint 1, 0xffffffff81c00ba0 in invalid_op () at arch/x86/entry/entry_64.S:1027
1027    idtentry bounds                         do_bounds                       has_error_code=0
(gdb) bt
#0  0xffffffff81c00ba0 in invalid_op () at arch/x86/entry/entry_64.S:1027
#1  0xffffffff810a54a1 in __disable_runtime (rq=<optimized out>) at kernel/sched/myp.c:430
#2  rq_offline_myp (rq=0xffff8881f9be7bc0) at kernel/sched/myp.c:1445
#3  0xffff8881f881d000 in ?? ()
#4  0x0000000000000286 in ?? ()
#5  0x000000000000000f in fixed_percpu_data ()
#6  0xffff8881f8031e00 in ?? ()
#7  0xffffffff81091b0c in set_rq_offline (rq=0xffff8881f9be8420) at kernel/sched/core.c:6356
#8  0xffffffff810a72da in rq_attach_root (rq=0xffff8881f9be7bc0, rd=0xffffffff82b13bb8 <def_root_domain+24>)
    at kernel/sched/topology.c:451
#9  0xffffffff810a73ca in cpu_attach_domain (sd=0xffff8881f8031e00, rd=<optimized out>, cpu=15)
    at kernel/sched/topology.c:699
#10 0xffffffff810a83e2 in build_sched_domains (cpu_map=0xffff8881f888a090, attr=<optimized out>)
    at kernel/sched/topology.c:2028
#11 0xffffffff810a8cf1 in sched_init_domains (cpu_map=<optimized out>) at kernel/sched/topology.c:2116
#12 0xffffffff829eb3c6 in sched_init_smp () at kernel/sched/core.c:6536
--Type <RET> for more, q to quit, c to continue without paging--
#13 0xffffffff829cf03d in kernel_init_freeable () at init/main.c:1185
#14 0xffffffff81ab59cf in kernel_init (unused=<optimized out>) at init/main.c:1109
#15 0xffffffff81c001a2 in ret_from_fork () at arch/x86/entry/entry_64.S:352
#16 0x0000000000000000 in ?? ()
```

从上面的堆栈可以看出，调用到 myp 调度器的__disable_runtime 函数时触发了异常，这个已经到了我们的函数，已经基本找到源头了，因此可以到 myp 的__disable_runtime 函数中去分析一下原因。

4.1.5. 跟踪分析__disable_runtime 函数的调用

```
Thread 1 hit Breakpoint 1, __disable_runtime (rq=<optimized out>) at kernel/sched/myp.c:1445
1445            __disable_runtime(rq);
(gdb) s
390             for_each_myp_rq(rt_rq, iter, rq) {
(gdb) n
395                     raw_spin_lock(&rt_b->rt_runtime_lock);
(gdb)
396                     raw_spin_lock(&rt_rq->rt_runtime_lock);
(gdb)
397                     if (0 == rt_rq->rt_runtime) {
(gdb)
400                     if (rt_rq->rt_runtime == RUNTIME_INF ||
(gdb)
403                     raw_spin_unlock(&rt_rq->rt_runtime_lock);
(gdb)
405                     want = rt_b->rt_runtime - rt_rq->rt_runtime;
(gdb)
408                             struct rt_rq *iter = sched_myp_period_myp_rq(rt_b, i);
(gdb) p want
$1 = -950000000
(gdb) n
429                     raw_spin_lock(&rt_rq->rt_runtime_lock);
(gdb)
430                     BUG_ON(want);
(gdb)
```

```
Thread 1 hit Breakpoint 1, __disable_runtime (rq=<optimized out>) at kernel/sched/myp.c:1445
1445            __disable_runtime(rq);
(gdb) s
390             for_each_myp_rq(rt_rq, iter, rq) {
(gdb) n
395                     raw_spin_lock(&rt_b->rt_runtime_lock);
(gdb)
396                     raw_spin_lock(&rt_rq->rt_runtime_lock);
(gdb)
397                     if (0 == rt_rq->rt_runtime) {
(gdb)
400                     if (rt_rq->rt_runtime == RUNTIME_INF ||
(gdb)
403                     raw_spin_unlock(&rt_rq->rt_runtime_lock);
(gdb)
405                     want = rt_b->rt_runtime - rt_rq->rt_runtime;
(gdb)
408                             struct rt_rq *iter = sched_myp_period_myp_rq(rt_b, i);
(gdb)
408                             struct rt_rq *iter = sched_myp_period_myp_rq(rt_b, i);
(gdb)
414                             raw_spin_lock(&iter->rt_runtime_lock);
(gdb)
415                             if (want > 0) {
(gdb) n
rq_offline_myp (rq=0xffff8881f9827bc0) at kernel/sched/myp.c:1445
1445            __disable_runtime(rq);
```

开始跟踪__disable_runtime，结果发现了如下异常：

```
397                     want = rt_b->rt_runtime - rt_rq->rt_runtime;
(gdb)
400                             struct rt_rq *iter = sched_myp_period_myp_rq(rt_b, i);
(gdb) p want
$3 = -950000000
(gdb) p rt_rq->rt_runtime
$4 = 950000000
(gdb) p def_myp_bandwidth->rt_runtime
$5 = 0
(gdb)
```

1）want都小于0了，不正常

2）不正常的原因是def_myp_bandwithd的rt_runtime=0

从上面的分析可以看到大概率是 def_myp_bandwidth 没有初始化导致。

基于这个分析，在 kernel/sched/core.c 中增加对 myp 调度器的初始化，如下：

init_myp_bandwidth(&def_myp_bandwidth,global_rt_period(), global_rt_runtime());

至此问题解决

4.2. 调试 shmem_init 函数无法访问的问题

实际上不只是 shmem_init 函数，内核的很多函数多无法访问了，发现这个问题是自己模仿内存文件系统裁剪了一个简单的文件系统，为了防止启动过程中死机，改成了修改文件系统节点是加载，就调用文件系统初始化函数，如：

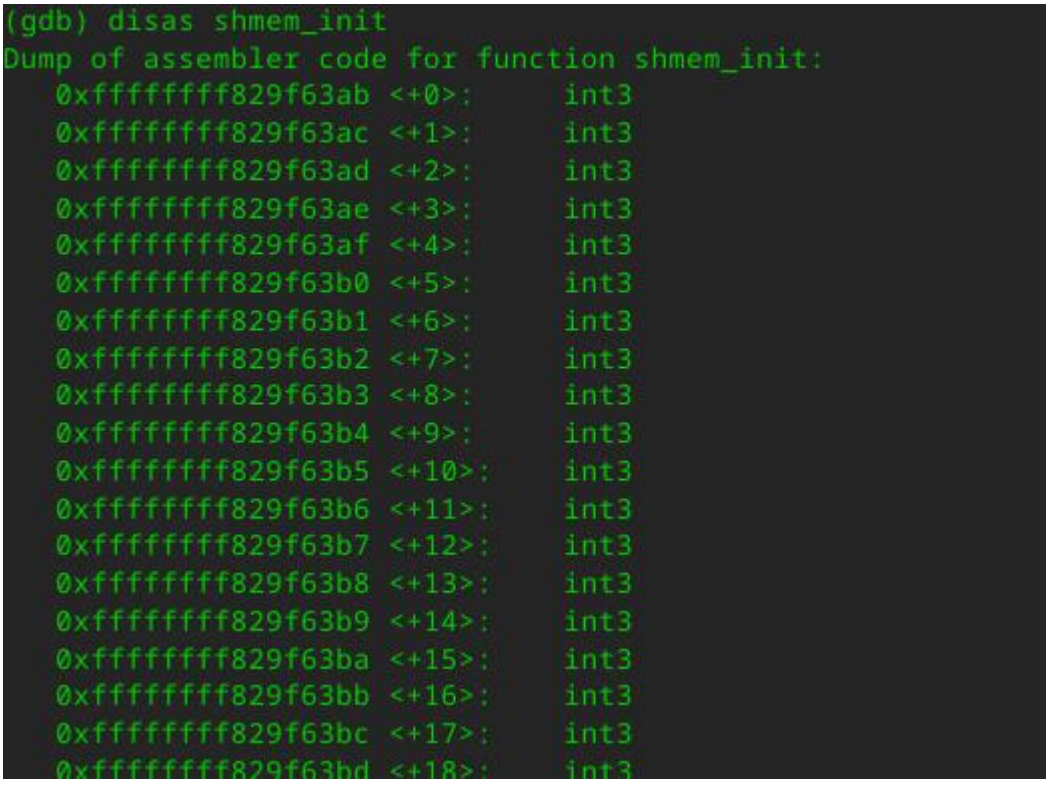
echo 10000 > /sys/kernel/profiling，就会触发 myfs_init 函数的调用， 结果调试过程中发现，只要一调用 myfs_init 就会死机，即使这个函数函数什么都不做，什么原因呢？基于虚拟机调试可以比较方便地找到问题的根因。

为了方便讲解，现在基于 shmem_init 来分析。

先看 x86 环境下的调试情况

4.2.1. x86 问题介绍

先看内核启动起来之后，基于 gdb 查看的 shmem_init 函数的情况



```
(gdb) x /4w shmem_init
0xffffffff829f63ab <shmem_init>:        0xcccccccc      0xcccccccc      0xcccccccc      0xcccccccc
(gdb)
```

这个函数对应的内存都变成了 0xCC，正常情况下的值是怎样的呢？

重新启动，hb start_kernel，当 start_kernel 函数被断下来后，查看 shmem_init 函数的内存值

```
(gdb) x /4w shmem_init
0xffffffff829f63ab <shmem_init>:        0xc0c74953      0x8117a2a0      0x040000b9      0xbed23100
(gdb)
```

再看看此时的汇编代码

```
(gdb) disas shmem_init
Dump of assembler code for function shmem_init:
   0xffffffff829f63ab <+0>:     push   %rbx
   0xffffffff829f63ac <+1>:     mov    $0xffffffff8117a2a0,%r8
   0xffffffff829f63b3 <+8>:     mov    $0x40000,%ecx
   0xffffffff829f63b8 <+13>:    xor    %edx,%edx
   0xffffffff829f63ba <+15>:    mov    $0x2a0,%esi
   0xffffffff829f63bf <+20>:    mov    $0xffffffff8220f9e7,%rdi
   0xffffffff829f63c6 <+27>:    callq  0xffffffff81186ba0 <kmem_cache_create>
   0xffffffff829f63cb <+32>:    mov    $0xffffffff8245e9a0,%rdi
```

结论：

shmem_init 函数的内存值是在内核启动完成后被释放了,那么是在哪个函数中释放的呢？释放的本质是什么呢？free pages 然后将内存值设置为 0xcc 吗？

arm 环境下又是怎样的呢？

4.2.2. arm 问题介绍

arm vm 启动起来后看到的现象如下：

```
(gdb) x /4w shmem_init
0xffff000009220e84 <shmem_init>:        Cannot access memory at address 0xffff000009220e84
(gdb) disas shmem_init
Dump of assembler code for function shmem_init:
   0xffff000009220e84 <+0>:     Cannot access memory at address 0xffff000009220e84
```

相对于 x86 环境，shmem_init 函数都无法访问了，就是说根据这个虚拟地址无法找到物理地址了，已经没有了内存映射，比 x86 上释放得更彻底。

总结：

arm 内核中除了释放内存，还解除了内存映射

需要找到是否内存和解除映射的地方，通过对比就可以知道两者处理的不同了

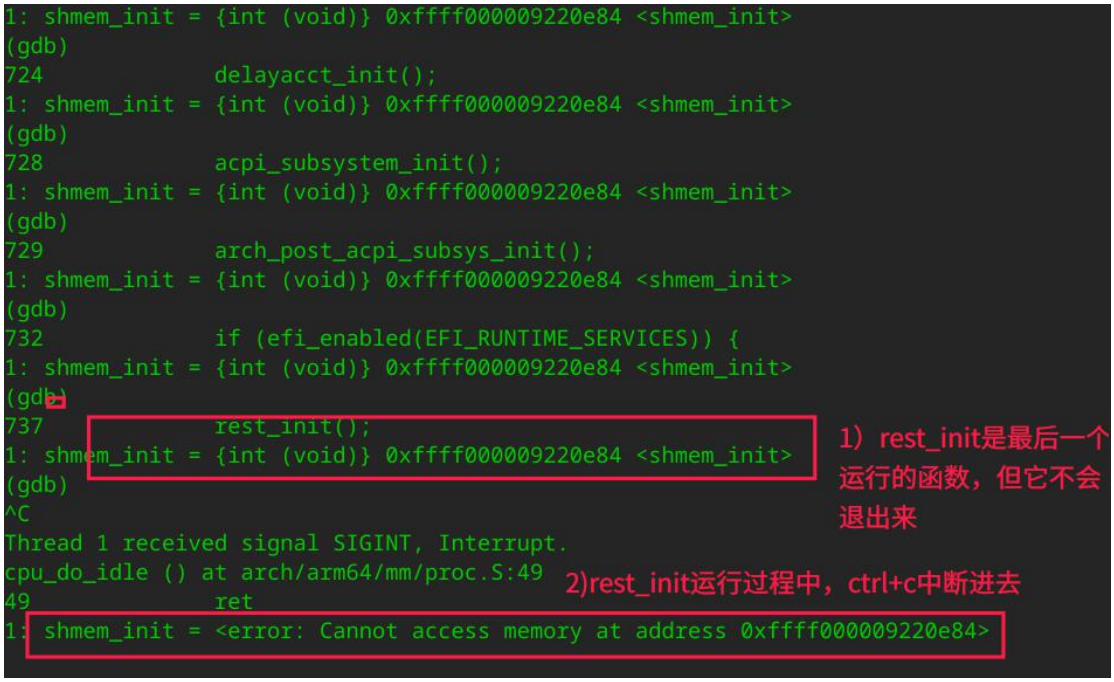
4.2.3. arm 环境调试

shmem_init 是代码，代码也需要被加入到内存，刚开启启动时 shmem_init 函数肯定被运行过，运行后内核为了防止重复运行，或者说为了优化内存，释放了不再需要运行的代码段，那么这个是否肯定发生在内核完成了初始化之后，可以基于内核的几个关键函数来

调试快速确认，如可以从 start_kernel 函数调试，看看调用了哪个函数就出现了访问不了的情况了。

b start_kernel
display shmem_init

单步跟踪结果如下：

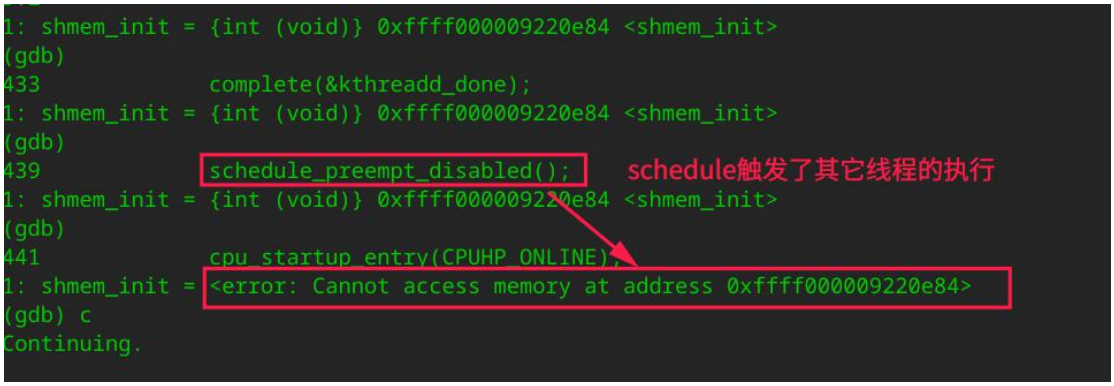


结论：

可以猜测内核代码段的释放发生在 rest_init 函数内部，需要进一步看看此函数内部调用了什么。

4.2.4. 分析 rest_init 函数

rest_int 函数继续单独跟踪，结果如下：



从上面的跟踪结果可以看到 schedule_preempt_disabled 执行完后 shmem_init 函数就无法访问了，而该函数内部主要是调用了 schedule 函数，schedule 函数的本质就是放弃运行，让其它线程去执行，因此可以猜测肯定是该函数中启动的线程运行后导致

shmem_init 函数无法访问。

rest_init 函数启动了两个线程：

pid = kernel_thread(kernel_init, NULL, CLONE_FS);

pid = kernel_thread(kthreadd, NULL, CLONE_FS | CLONE_FILES);

kernel_init 完成内核的初始化，kthreadd 负责内核线程的创建。

因此很有可能发生在 kernel_init 函数内部

4.2.5. 调试 kernel_init 函数

单步跟踪 kernel_init 函数，很快发现了触发者，如下：

可以明确 free_initmem 函数运行后，shmem_init 函数就不能访问，现在来看看它里面的代码：

从上面的代码可以看到，内核完成初始化后调用 free_initmem 释放了不会在调用的代码占用的内存空间，且解除了虚拟地址到物理地址的映射。

当 free_reserved_area 函数调用后，arm 版本上，内存已经被清 0 并释放了，此时反汇编会显示 undefined。此时根据虚拟地址可以找到物理地址，虽然物理地址已经不属于该虚拟地址了，但 mmu 的计算算法就是可以找到它。

当 unmap_kernel_range 调用后，pte 页表已经清空，无法根据虚拟地址无法访问到

物理地址了，因此此时访问此虚拟地址，就会显示不可访问了。

根据__init_begin 和__init_end 的地址可以知道被释放的内存空间，在我的调试环境中发现这个内存空间的大小是[0xffff000009200000, 0xffff000009800000], 6M 的内存空间，只要是这个内存空间的代码无法再看到它的汇编代码，触发屏蔽掉对 free_initmem 的调用。

4.2.6. x86 free_initmem 函数确认

为什么 arm 上函数已经无法访问，但是 x86 上还能访问只是内存内容被设置成了 0xCC 呢?

这个差异就在于 free_initmem 的实现不一样, x86 的适配代码仅仅释放了内存

a)对于这个问题本身，我们得到的经验就是因为内存被释放了，且 arm 版本中还将 pte 页表也是释放了。我们用 gdb 能调试内核的强大功能给出了一条快速找到释放代码的方法。

b)如果以后遇到了运行一段时间后就出现不能再访问的问题,可以基于条件断点跟踪释放函数，基于虚拟地址跟踪释放页表函数 vunmap_page_range

c)哪些函数在内核启动后就不再存在可以根据 cat /proc/kallsyms 展现出来的函数地址和__init_begin、__init_end 的地址范围快速得知，arm 上这个释放的空间有 6M

d)以后调试过程中反汇编提示 undefined 的错误时需要马上想到不是此函数没有实现，而是内存被释放了。同时也不要尝试在运行过程中调用这些初始化才应该调用的函数，否则就会出现死机。

5. 总结

a) 内核非常庞大，bpf 可以帮助我们调试运行过程中的调用，但是它只能调试部分函数，无法调试启动过程，更无法跟踪函数内部的细节，这些虚拟机调试都可以解决。

b) 更重要的是，内核稍有 bug 就会导致系统无法开机， 编译 deb 格式的内核包也要比值生成 vmlinux 和 bzImage 慢很多，因此重新启动调试的效率也有很大区别

c) 本文中的具体例子都是为了展示如何灵活地基于 gdb 去调试，如实现自己的驱动触发内核相关函数的调用；实现自己的函数，方便 call 调用；将变量的值 snprintf 到一个缓冲区方便 gdb print 出来；基于 SP_EL0 寄存器的值直接获取当前运行进程的 task_struct 等，都是根据自己的调试需求去灵活调试

d) x86 和 arm 调试环境都提供了内核，可以直接 make -j32 方式编译，存在编译内核时需要工具没有安装的情况，需要自己安装；x86 虚拟机调试设置断点需要调用 hb 命令设置硬件断点，x86 的根文件系统比较强大，在目标机器 share 中的文件在虚拟机环境中都可以查看和使用。arm 调试环境比较简洁，包也更小，每次修改需要重新调用 busybox 下面的 build-pkg 生成根文件系统。