KDUMP+CRASH解决产品研发和云服务器中的死机难题

笨叔叔 奔跑吧Linux社区

Gavin Guo Ubuntu

目 录

kdump+crash简介

为什么需要kdump?

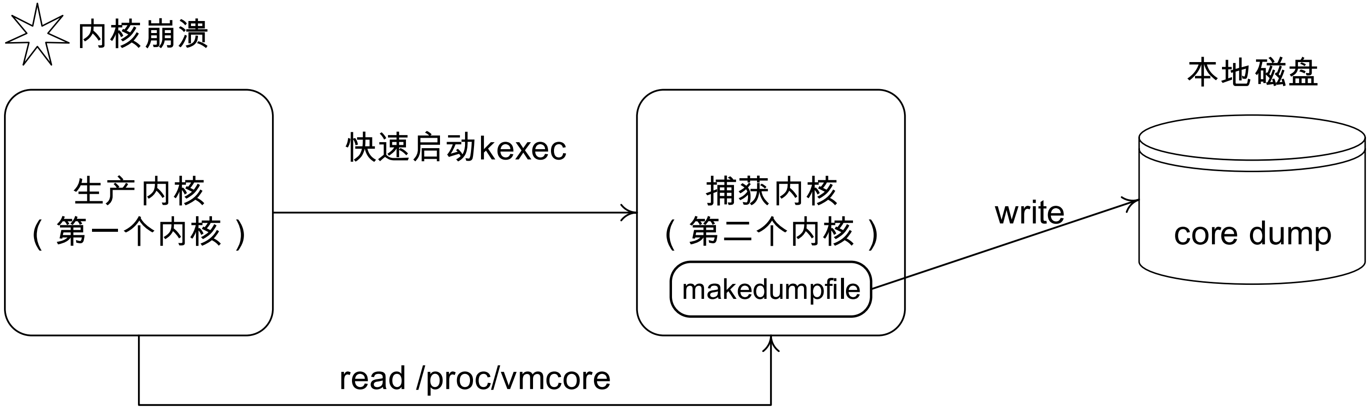
- 发展了28年的Linux内核 真的很健壮吗?
- 是打不死的小强还是弱不禁风的女子?

- 一个不起眼驱动里的空指针访问可以让Linux系统panic!

什么是kdump+crash?

- what? ：发生崩溃时刻的系统内存的一个快照
- who? 系统运维（服务器运维），产品底层开发人员（BSP），产品性能优化
- where? 使用Linux内核的物理机器和虚拟机
- when? 当你的系统没有响应（unresponsive）时。kdump主要是用来分析系统死机黑屏无响应等问题。若是硬件问题导致的死机，特别是不能重新热启动的，基本kdump无能为力了。
- why? 研究为啥Linux内核发生了崩溃
- how? 部署kdump+crash，并学习如何使用它

kdump工作原理


内核崩溃
生产内核
（第一个内核）
快速启动kexec
捕获内核
（第二个内核）
makedumpfile
write
本地磁盘
core dump
read /proc/vmcore

咋触发一个kdump？

- kdump通常用于系统假死机（unresposive）和panic，也就是没有响应的情况。硬件问题导致的直接死机，kdump无能为力！

- kdump触发条件
- 手动触发：
  - sysrq
  - NMI
- 自动触发
  - kernel panic
  - watchdog
  - Hard/Soft lockup
  - out of memory

我们常常遇到被block住的进程，怎么办?

```
[  484.400909] INFO: task ps:1278 blocked for more than 120 seconds.
[  484.401355]       Tainted: G           OE     5.0.0+ #1
[  484.401738] "echo 0 > /proc/sys/kernel/hung_task_timeout_secs" disables this message.
[  484.402255] ps              D    0  1278    550 0x00000001
[  484.402598] Call trace:
[  484.402763]  __switch_to+0xbf4/0xc24
[  484.403000]  __schedule+0x1760/0x1858
[  484.403207]  schedule+0x2ac/0x35c
[  484.403775]  rwsem_down_read_failed+0xb54/0xe40
[  484.404106]  down_read+0x90/0x1ac
[  484.404378]  __access_remote_vm+0x60/0x3a4
[  484.404659]  access_remote_vm+0x44/0x4c
[  484.404934]  get_mm_cmdline+0x480/0x810
[  484.405222]  get_task_cmdline+0x58/0x70
[  484.405526]  proc_pid_cmdline_read+0xb0/0x140
[  484.405877]  __vfs_read+0x58/0x94
[  484.406076]  vfs_read+0x120/0x248
[  484.406316]  ksys_read+0xb4/0x150
[  484.406550]  __se_sys_read+0x4c/0x5c
[  484.406827]  __arm64_sys_read+0x44/0x4c
[  484.407119]  __invoke_syscall+0x28/0x30
[  484.407735]  invoke_syscall+0xa8/0xdc
[  484.408033]  el0_svc_common+0xf8/0x1d4
[  484.408286]  el0_svc_handler+0x3bc/0x3e8
[  484.408579]  el0_svc+0x8/0xc
```

部署kdump：x86系统服务器 – Ubuntu server

Download Ubuntu Server

Ubuntu Server 18.04.2 LTS

The long-term support version of Ubuntu Server, including the Queens release of OpenStack and support guaranteed until April 2023 — 64-bit only.

This release uses our new installer, Subiquity. If you need support for options not implemented in Subiquity, such as encrypted filesystem support, the traditional installer can be found on the alternative downloads page.

Ubuntu Server 18.04 LTS release notes

Download

For other versions of Ubuntu including torrents, the network installer, a list of local mirrors, and past releases see our alternative downloads.

Ubuntu Server 坚如磐石

部署kdump：arm64实验平台

- 笨叔制作了一个Linux 5.0 + kdump + arm64的实验平台
- git clone https://github.com/figozhang/linux-5.0-kdump.git
- 如何进行kdump实验：阅读里面README
- 采用Linux 5.0内核 + debian rootfs制作，在QEMU下运行。

部署kdump：嵌入式平台

- 内核需要配置如下选项：
  - CONFIG_KEXEC=y
  - CONFIG_CRASH_DUMP=y

- 下载kexec最新源码包，（交叉）编译安装。

- 启动内核配置capture内核预留内存：比如：crashkernel=256M

- 启动kexec命令：
  - ✓ 比如：kexec -p --command-line="root=/dev/vda rw" /boot/Image
  - ✓ 写转存文件：cp /proc/vmcore /var/crash/dump.core

简单上手 – 手动触发一个dump

- 检查kdump服务

```
root@benshushu:~# systemctl status kdump-tools
● kdump-tools.service - Kernel crash dump capture service
   Loaded: loaded (/lib/systemd/system/kdump-tools.service; enabled; vendor pres
   Active: active (exited) since Wed 2019-05-29 14:25:20 UTC; 1min 12s ago
  Process: 283 ExecStart=/etc/init.d/kdump-tools start (code=exited, status=0/SU
 Main PID: 283 (code=exited, status=0/SUCCESS)

May 29 14:25:12 benshushu systemd[1]: Starting Kernel crash dump capture service
May 29 14:25:16 benshushu kdump-tools[283]: Starting kdump-tools: Creating symli
May 29 14:25:16 benshushu kdump-tools[283]: Creating symlink /var/lib/kdump/init
May 29 14:25:19 benshushu kdump-tools[283]: loaded kdump kernel.
May 29 14:25:20 benshushu systemd[1]: Started Kernel crash dump capture service.
```

- 手动触发一个panic和dump: echo c > /proc/sysrq-trigger

```
root@benshushu:~# echo c > /proc/sysrq-trigger
[  622.650160] sysrq: SysRq : Trigger a crash
[  622.652927] Kernel panic - not syncing: sysrq triggered crash
[  622.653561] CPU: 1 PID: 694 Comm: bash Kdump: loaded Tainted: G            E     5.0.0-rlk+ #1
[  622.653892] Hardware name: linux,dummy-virt (DT)
[  622.654224] Call trace:
[  622.654868]  dump_backtrace+0x0/0x1b0
[  622.655035]  show_stack+0x24/0x30
[  622.655175]  dump_stack+0x90/0xb4
[  622.655310]  panic+0x144/0x310
[  622.655439]  sysrq_handle_crash+0x1c/0x20
[  622.655593]  __handle_sysrq+0xac/0x198
[  622.655783]  write_sysrq_trigger+0x70/0x88
[  622.655943]  proc_reg_write+0x78/0xc8
[  622.656086]  __vfs_write+0x60/0x1a8
[  622.656222]  vfs_write+0xac/0x1b8
[  622.656351]  ksys_write+0x6c/0xd8
[  622.656481]  __arm64_sys_write+0x24/0x30
[  622.656629]  el0_svc_common+0x78/0x120
[  622.656771]  el0_svc_handler+0x38/0x78
[  622.656912]  el0_svc+0x8/0xc
[  622.657718] SMP: stopping secondary CPUs
[  622.659424] Starting crashdump kernel...
[  622.659850] Bye!
```

```
         Starting Kernel crash dump capture service...
[   26.460080] input: gpio-keys as /devices/platform/gpio-keys/input/input0
[   27.543369] kdump-tools[234]: Starting kdump-tools: running makedumpfile -c -d 31 /proc/vmcore /var/crash/201906221945/
dump-incomplete.
Copying data                                       : [  0.0 %] -
```

```
      KERNEL: /mnt/vmlinux
    DUMPFILE: vmcore.201906221945
        CPUS: 4
        DATE: Sat Jun 22 19:45:17 2019
      UPTIME: 00:18:26
LOAD AVERAGE: 0.00, 0.05, 0.06
       TASKS: 88
    NODENAME: benshushu
     RELEASE: 5.0.0-rlk+
     VERSION: #1 SMP Thu Jun 20 05:53:19 CST 2019
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 1 GB
       PANIC: "sysrq: SysRq : Trigger a crash"
         PID: 694
     COMMAND: "bash"
        TASK: ffff800023da3a00  [THREAD_INFO: ffff800023da3a00]
         CPU: 1
       STATE: TASK_RUNNING (SYSRQ)

crash>
```

分析dump数据的武器：crash工具

- 基于GDB的一个分析工具，由红帽工程师开发和维护
- crash常用命令

```
crash> help

*              extend         log            rd             task
alias          files          mach           repeat         timer
ascii          foreach        mod            runq           tree
bpf            fuser          mount          search         union
bt             gdb            net            set            vm
btop           help           p              sig            vtop
dev            ipcs           ps             struct         waitq
dis            irq            pte            swap           whatis
eval           kmem           ptob           sym            wr
exit           list           ptov           sys            q
```

分析dump数据的武器：crash工具

- 基于GDB的一个分析工具，由红帽工程师开发和维护
- crash常用命令

```
crash> help

*              extend         log            rd             task
alias          files          mach           repeat         timer
ascii          foreach        mod            runq           tree
bpf            fuser          mount          search         union
bt             gdb            net            set            vm
btop           help           p              sig            vtop
dev            ipcs           ps             struct         waitq
dis            irq            pte            swap           whatis
eval           kmem           ptob           sym            wr
exit           list           ptov           sys            q
```

实战案例

九大死机实战案例

实战1： 简单的oops错误
实战2：访问了已经删除的链表节点
实战3： 内存访问bug
实战4：实战驱动的死机问题
实战5：小牛试刀死锁
实战6：恢复函数调用栈
实战7：分析和推导复杂的变量
实战8：分析和解决一个复杂的死锁问题
实战9：企业级虚拟化计算节点服务器性能问题

实战1: 分析简单的oops错误

1. 通过本实验了解crash工具如何使用
2. 通过本实验了解crash大致如何分析问题

```c
struct mydev_priv {
	char name[64];
	int i;
};
```

```c
int create_oops(struct vm_area_struct *vma, struct mydev_priv *priv)
{
	unsigned long flags;

	flags = vma->vm_flags;
	printk("flags=0x%lx, name=%s\n", flags, priv->name);

	return 0;
}

int __init my_oops_init(void)
{
	int ret;
	struct vm_area_struct *vma = NULL;
	struct mydev_priv priv;

	vma = kmalloc(sizeof (*vma), GFP_KERNEL);
	if (!vma)
		return -ENOMEM;

	kfree(vma);
	vma = NULL;

	smp_mb();

	memcpy(priv.name, "figo", sizeof("figo"));
	priv.i = 10;

	ret = create_oops(vma, &priv);

```

```
      KERNEL: /mnt/vmlinux
    DUMPFILE: dump.201906201406  [PARTIAL DUMP]
        CPUS: 4
        DATE: Thu Jun 20 14:05:32 2019
      UPTIME: 00:34:51
LOAD AVERAGE: 0.39, 0.22, 0.26
       TASKS: 87
    NODENAME: benshushu
     RELEASE: 5.0.0-rlk+
     VERSION: #1 SMP Thu Jun 20 05:53:19 CST 2019
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 1 GB
       PANIC: "Unable to handle kernel NULL pointer dereference at virtual address 0000000000000050"
         PID: 1659
     COMMAND: "insmod"
        TASK: ffff800026b78000  [THREAD_INFO: ffff800026b78000]
         CPU: 3
       STATE: TASK_RUNNING (PANIC)

crash>
```

分析函数调用栈信息

```
crash> bt
PID: 1247   TASK: ffff80009a46aac0  CPU: 2   COMMAND: "insmod"
 #0 [ffff00000b903520] machine_kexec at ffff00000809ffe4
 #1 [ffff00000b903580] __crash_kexec at ffff000008195734
 #2 [ffff00000b903710] crash_kexec at ffff000008195844
 #3 [ffff00000b903740] die at ffff00000808e63c
 #4 [ffff00000b903780] die_kernel_fault at ffff0000080a3d0c
 #5 [ffff00000b9037b0] __do_kernel_fault at ffff0000080a3dac
 #6 [ffff00000b9037e0] do_page_fault at ffff0000088ee49c
 #7 [ffff00000b9038e0] do_translation_fault at ffff0000088ee7c8
 #8 [ffff00000b903910] do_mem_abort at ffff000008081514
 #9 [ffff00000b903b10] el1_ia at ffff00000808318c
     PC: ffff000000e54020  [create_oops+32]
     LR: ffff000000e590a0  [_MODULE_INIT_START_oops+160]
     SP: ffff00000b903b20  PSTATE: 80000005
    X29: ffff00000b903b20  X28: ffff000008b67000  X27: ffff000000e56180
    X26: ffff00000b903dc0  X25: ffff000000e56198  X24: ffff000000e56008
    X23: 0000000000000000  X22: ffff000000e56000  X21: ffff000009089708
    X20: ffff000000e59000  X19: ffff000000e56000  X18: 0000000000000000
    X17: 0000000000000000  X16: 0000000000000000  X15: ffffffffffffffff
    X14: ffff000009089708  X13: 0000000000000040  X12: 0000000000000228
    X11: 0000000000000000  X10: 0000000000000000   X9: 0000000000000001
     X8: ffff8000ba0fc900   X7: ffff8000bae73b00   X6: ffff00000b903b89
     X5: 00000000000008a6   X4: ffff8000bb6b9b40   X3: 0000000000000000
     X2: ffff000000e590a0   X1: ffff00000b903b84   X0: 0000000000000000
#10 [ffff00000b903b20] create_oops at ffff000000e5401c [oops]
#11 [ffff00000b903b50] _MODULE_INIT_START_oops at ffff000000e5909c [oops]
#12 [ffff00000b903bd0] do_one_initcall at ffff000008084868
#13 [ffff00000b903c60] do_init_module at ffff00000818f964
#14 [ffff00000b903c90] load_module at ffff0000081917f0
#15 [ffff00000b903d80] __se_sys_finit_module at ffff000008191acc
#16 [ffff00000b903e40] __arm64_sys_finit_module at ffff000008191d90
#17 [ffff00000b903e60] el0_svc_common at ffff000008096a10
```

造成内核崩溃的进程

造成内核崩溃的指令

堆栈指针寄存器sp

栈帧基地址寄存器fp

传递函数参数1

传递函数参数2

内核态函数调用关系

1. 使用mod命令加载带符号信息的内核模块。

crash> mod -s oops /home/benshushu/crash/crash_lab_arm64/01_oops/oops.ko
MODULE NAME SIZE OBJECT FILE
ffff000000e56000 oops 16384 /home/benshushu/crash/crash_lab_arm64/01_oops/oops.ko
crash>

2. 反汇编PC寄存器指向的地方，也就是内核崩溃发生的地方。

crash> dis -l ffff000000e54020
0xffff000000e54020 <create_oops+32>: ldr x0, [x0,#80]
crash>

```
crash> struct -o vm_area_struct
struct vm_area_struct {
    [0] unsigned long vm_start;
    [8] unsigned long vm_end;
   [16] struct vm_area_struct *vm_next;
   [24] struct vm_area_struct *vm_prev;
   [32] struct rb_node vm_rb;
   [56] unsigned long rb_subtree_gap;
   [64] struct mm_struct *vm_mm;
   [72] pgprot_t vm_page_prot;
   [80] unsigned long vm_flags;
```

3. *查看数据结构的偏移量*

crash> struct -o vm_area_struct
struct vm_area_struct {
[0] unsigned long vm_start;
...
[80] unsigned long vm_flags;

4. *struct命令查看数据结构的值*

crash> struct vm_area_struct 0x0
struct: invalid kernel virtual address: 0x0
crash>

```
crash> struct vm_area_struct 0x0
struct: invalid kernel virtual address: 0x0
crash>
```

查看第二参数的内容

```
crash> rd ffff00000b903b84
ffff00000b903b84:  0b903d006f676966                   figo.=..
crash> struct mydev_priv ffff00000b903b84
struct mydev_priv {
  name = "figo\000=\220\v\000\000\377\377\200a\345\000\000\000\37
0\000\000\271\236\233\000\200\377\377\000\000\000\000\000\000\000
  i = 10
}
crash> 
```

实验2:访问了已经删除的链表节点

创建三个线程，一个是添加list，另外是remove list，最后一个是删除list一个元素

```c
static int list_add_thread(void *p)
{
        int i;

        while (!kthread_should_stop()) {
                spin_lock(&lock);
                for (i = 0; i < 1000; i++) {
                        struct foo *new_ptr = kmalloc(sizeof (struct foo), GFP_ATOMIC);
                        new_ptr->a = i;
                        list_add_tail(&new_ptr->list, &g_test_list);
                }
                spin_unlock(&lock);
                msleep(20);
        }

        return 0;
}
```

```c
static int list_del_thread(void *data)
{
        struct foo *entry;

        while (!kthread_should_stop()) {
                if (!list_empty(&g_test_list)) {
                        spin_lock(&lock);
                        entry = list_entry(g_test_list.next, struct foo, list);
                        list_del(&entry->list);
                        //kfree(entry);
                        spin_unlock(&lock);
                }
                msleep(1);
        }

        return 0;
}
```

```c
static int list_remove_thread(void *data)
{
        struct foo *entry;

        while (!kthread_should_stop()) {
                spin_lock(&lock);
                while (!list_empty(&g_test_list)) {
                        entry = list_entry(g_test_list.next, struct foo, list);
                        list_del(&entry->list);
                        kfree(entry);
                }
                spin_unlock(&lock);
                mdelay(10);
        }

        return 0;
}
```

实战3: 内存访问bug

```c
static void mem_timefunc(unsigned long dummy)
{
        struct vm_area_struct *vma = (struct vm_area_struct *)(dummy);

        printk("%s: set vma = %p\n", __func__, vma);

        vma->vm_flags = 1;
        vma->vm_pgoff = 1;
}

int create_oops(struct vm_area_struct **p)
{
        unsigned long flags;
        struct vm_area_struct *vma = *p;

        flags = vma->vm_flags;
        printk("flags=0x%lx\n", flags);
        printk("%s: free vma %p\n", __func__, vma);

        kfree(*p);
        *p = NULL;

        return 0;
}

```

```c
static int __init my_oops_init(void)
{
        int ret;

        gvma = kmalloc(sizeof (*gvma), GFP_ATOMIC);
        if (!gvma)
                return -ENOMEM;

        printk("%s, gvma=%p\n", __func__, gvma);

        ret = create_oops(&gvma);

        timer.expires = jiffies + msecs_to_jiffies(10);
        setup_timer(&timer, mem_timefunc, (unsigned long)gvma);
        add_timer(&timer);

        return 0;
}
```

实验4: 实战驱动的死机问题

我们写驱动程序，涉及到寄存器的读写，内核提供了一个regmap的framework，我们可以使用regmap来管理系统中各种各样的寄存器读写的问题。下面笨叔写一个驱动来模拟一下寄存器读写：

```c
static void mem_timefunc(unsigned long dummy)
{
        struct vm_area_struct *vma = (struct vm_area_struct *)(dummy);

        printk("%s: set vma = %p\n", __func__, vma);

        vma->vm_flags = 1;
        vma->vm_pgoff = 1;
}

int create_oops(struct vm_area_struct **p)
{
        unsigned long flags;
        struct vm_area_struct *vma = *p;

        flags = vma->vm_flags;
        printk("flags=0x%lx\n", flags);
        printk("%s: free vma %p\n", __func__, vma);

        kfree(*p);
        *p = NULL;

        return 0;
}

```

```c
static int __init my_oops_init(void)
{
        int ret;

        gvma = kmalloc(sizeof (*gvma), GFP_ATOMIC);
        if (!gvma)
                return -ENOMEM;

        printk("%s, gvma=%p\n", __func__, gvma);

        ret = create_oops(&gvma);

        timer.expires = jiffies + msecs_to_jiffies(10);
        setup_timer(&timer, mem_timefunc, (unsigned long)gvma);
        add_timer(&timer);

        return 0;
}
```

实战5：小牛试刀死锁

在实验4基础上增加死锁

```c
static int _reg_read(void *context, unsigned int reg,
                     unsigned int *val)
{
        void __iomem *base = context;
        unsigned int status;

        printk("%s: reg=0x%x\n", __func__, reg);

        status = readl(base + REG_STATUS);

        while (status != 0xab) {
                cpu_relax();
                status = readl(base + REG_STATUS);
        }

        *val = readl(base + reg);

        printk("%s: reg=0x%x, val=0x%x\n", __func__, reg, *val);

          return 0;
}
```

实战5：小牛试刀死锁

在实验4基础上增加死锁

```
static int _reg_read(void *context, unsigned int reg,
                        unsigned int *val)
{
        void __iomem *base = context;
        unsigned int status;

        printk("%s: reg=0x%x\n", __func__, reg);

        status = readl(base + REG_STATUS);

        while (status != 0xab) {
                cpu_relax();
                status = readl(base + REG_STATUS);
        }

        *val = readl(base + reg);

        printk("%s: reg=0x%x, val=0x%x\n", __func__, reg, *val);

          return 0;
}
```

实战6：恢复函数调用栈

实验目的：理解arm64的栈是怎么布局的

已知Linux系统在奔溃时候的寄存器的值，请恢复函数调用关系图以及函数调用栈

```
 PC: ffff000000e54020  [create_oops+32]
 LR: ffff000000e590a0  [_MODULE_INIT_START_oops+160]
 SP: ffff00000b903b20  PSTATE: 80000005
X29: ffff00000b903b20  X28: ffff000008b67000  X27: ffff000000e56180
X26: ffff00000b903dc0  X25: ffff000000e56198  X24: ffff000000e56008
X23: 0000000000000000  X22: ffff000000e56000  X21: ffff000009089708
X20: ffff000000e59000  X19: ffff000000e56000  X18: 0000000000000000
X17: 0000000000000000  X16: 0000000000000000  X15: ffffffffffffffff
X14: ffff000009089708  X13: 0000000000000040  X12: 0000000000000228
X11: 0000000000000000  X10: 0000000000000000   X9: 0000000000000001
 X8: ffff8000ba0fc900   X7: ffff8000bae73b00   X6: ffff00000b903b89
 X5: 00000000000008a6   X4: ffff8000bb6b9b40   X3: 0000000000000000
 X2: ffff000000e590a0   X1: ffff00000b903b84   X0: 0000000000000000
```

已知条件如上图的30多个通用寄存器的值，
求解函数调用关系以及调用栈？

ARM

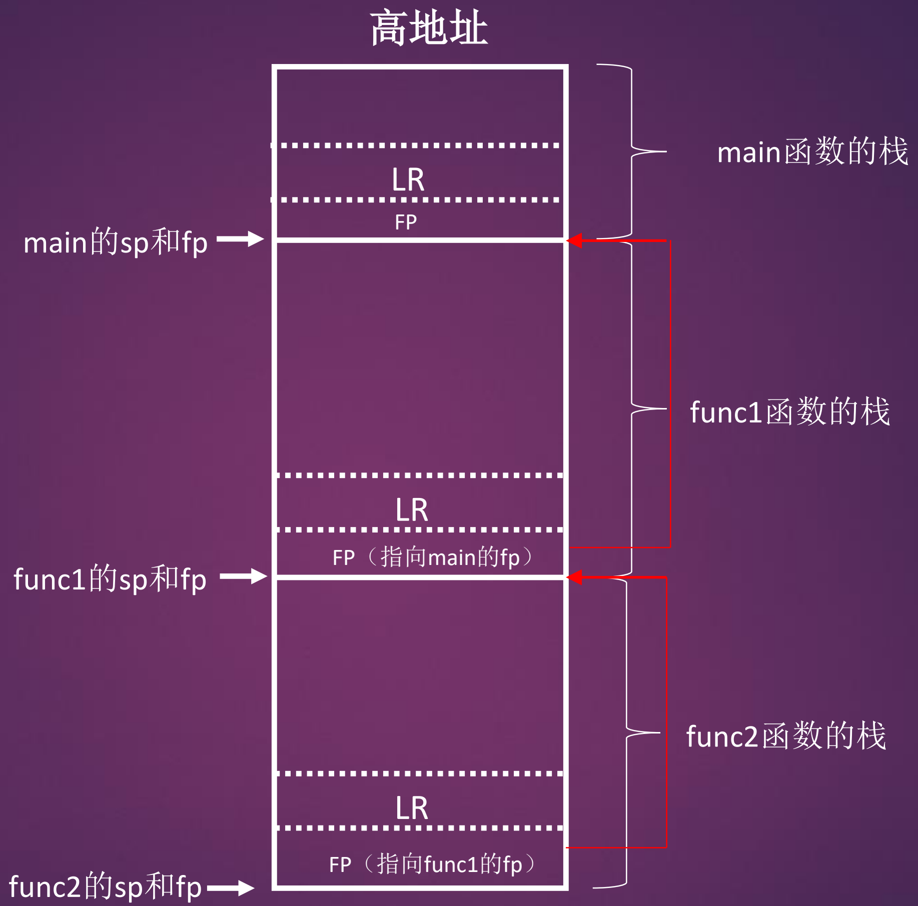
Procedure Call Standard for the ARM 64-bit Architecture (AArch64)

Document number: ARM IHI 0055B, current through AArch64 ABI release 1.0
Date of Issue: 22nd May 2013

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19…r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9…r15		Temporary registers
r8		Indirect result location register
r0…r7		Parameter/result registers

Table 2, General purpose registers and AAPCS64 usage

假设函数调用关系为：
main() -> func1() -> func2()



实战7：分析和推导复杂的变量

实验目的：

- ✓ 本章通过一个简单的实验来分析和推导复杂的变量
- ✓ 理解arm64的栈是怎么布局的
- ✓ 如何结合反汇编来推导出形参的值

实验前准备

使能panic

echo 1 > /proc/sys/kernel/softlockup_panic

echo 1 > /proc/sys/kernel/hung_task_panic

echo 60 > /proc/sys/kernel/hung_task_timeout_secs   //笨叔这里设置短一点

```
root@debian:/var/crash/lab1# sysctl -a | grep panic
kernel.hung_task_panic = 1
kernel.panic = 0
kernel.panic_on_oops = 1
kernel.panic_on_rcu_stall = 0
kernel.panic_on_warn = 0
kernel.softlockup_panic = 1
vm.panic_on_oom = 0
root@debian:/var/crash/lab1# 
```

```c
struct mydev_priv {
	char name[64];
	int i;
	struct mm_struct *mm;
	struct rw_semaphore *sem;
};
```

```c
int __init my_oops_init(void)
{
	int ret;
	struct vm_area_struct *vma = NULL;
	struct mydev_priv priv;
	struct mm_struct *mm;

	mm = get_task_mm(current);

	priv.mm = mm;
	priv.sem = &mm->mmap_sem;

	down_write(&mm->mmap_sem);

	vma = kmalloc(sizeof (*vma), GFP_KERNEL);
	if (!vma)
		return -ENOMEM;

	kfree(vma);
	vma = NULL;

	smp_mb();

	memcpy(priv.name, "figo", sizeof("figo"));
	priv.i = 10;

	ret = create_oops(vma, &priv, &mm->mmap_sem);

	return 0;
```

本实验的目的是分析出create_oops函数里面的第二个和第三个参数具体传递的是什么值?

```
      KERNEL: /mnt/vmlinux
    DUMPFILE: dump.201906230911  [PARTIAL DUMP]
        CPUS: 4
        DATE: Sun Jun 23 09:11:19 2019
      UPTIME: 00:32:18
LOAD AVERAGE: 0.73, 0.31, 0.17
       TASKS: 86
    NODENAME: benshushu
     RELEASE: 5.0.0-rlk+
     VERSION: #1 SMP Thu Jun 20 05:53:19 CST 2019
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 1 GB
       PANIC: "Kernel panic - not syncing: hung_task: blocked tasks"
         PID: 35
     COMMAND: "khungtaskd"
        TASK: ffff80002a0d1d00  [THREAD_INFO: ffff80002a0d1d00]
         CPU: 3
       STATE: TASK_RUNNING (PANIC)

crash>
```

崩溃的进程的PID

崩溃的原因

ps命令来查看UNINTERRIBLE的进程

```
crash> ps | grep UN
   1714    715   2  ffff8000b935c740  UN    0.0     2704    1540  insmod
crash> 
```

```
crash> bt 1714
PID: 1714   TASK: ffff8000b935c740  CPU: 2   COMMAND: "insmod"
 #0 [ffff00000c49b980] __switch_to at ffff000008087e78
 #1 [ffff00000c49b9a0] __schedule at ffff0000088e73b4
 #2 [ffff00000c49ba30] schedule at ffff0000088e79ec
 #3 [ffff00000c49ba40] rwsem_down_read_failed at ffff0000088eaeb8
 #4 [ffff00000c49bad0] down_read at ffff0000088ea450
 #5 [ffff00000c49baf0] create_oops at ffff000000e4f024 [oops]
 #6 [ffff00000c49bb30] _MODULE_INIT_START_oops at ffff000000e540dc [oops]
 #7 [ffff00000c49bbd0] do_one_initcall at ffff000008084868
 #8 [ffff00000c49bc60] do_init_module at ffff00000818f964
 #9 [ffff00000c49bc90] load_module at ffff0000081917f0
#10 [ffff00000c49bd80] __se_sys_finit_module at ffff000008191acc
#11 [ffff00000c49be40] __arm64_sys_finit_module at ffff000008191d90
#12 [ffff00000c49be60] el0_svc_common at ffff000008096a10
#13 [ffff00000c49bea0] el0_svc_handler at ffff000008096abc
#14 [ffff00000c49bff0] el0_svc at ffff000008084044
     PC: 0000ffff95fe6c34   LR: 0000aaaacc8a6140   SP: 0000ffff8794810
    X29: 0000ffff8794810  X28: 0000000000000000  X27: 0000000000000000
    X26: 0000000000000002  X25: 0000000000000000  X24: 0000ffff87948e8
    X23: 0000aaab0ac82760  X22: 0000000000000000  X21: 0000000000000000
    X20: 0000aaaacc8b0908  X19: 0000aaab0ac827a0  X18: 000000000000002f
    X17: 0000ffff95fe6c10  X16: 0000aaaacc8c7f50  X15: 0000000000000002
```

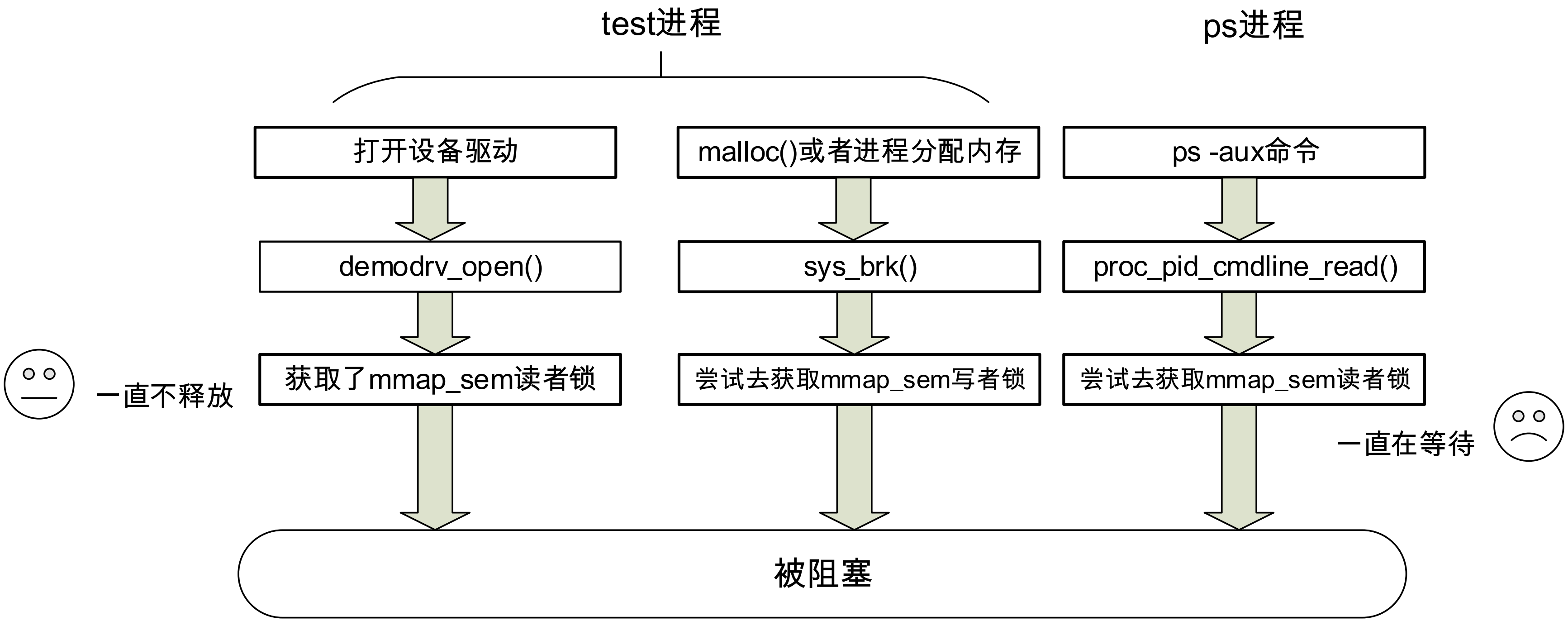
1. ps命令查看当前系统有哪些进程被block住了？

2. 查看1714进程的backtrace
3. 我们重点关注do_one_initcall->xxx_oops函数的调用关系
4. 通过这些栈的调用关系来找到create_oops函数的第2，和第3个参数的值

实战8：分析一个复杂的死锁例子

实验目的：

- 通过一个复杂的例子来学会如何利用crash工具来分析
- 如何通过栈来获取形参或者局部变量的值
- 如何分析和推导出：谁持有了这个锁?
- 如何分析和推导：哪些进程在睡眠等待这个锁?
- 通过实验深入理解读写信号量和mutex锁的原理和机制
- 通过本实验具有分析和解决服务器或者云服务以及嵌入式系统线上死机问题的能力


test进程
ps进程
打开设备驱动
malloc()或者进程分配内存
ps -aux命令
demodrv_open()
sys_brk()
proc_pid_cmdline_read()
一直不释放
获取了mmap_sem读者锁
尝试去获取mmap_sem写者锁
尝试去获取mmap_sem读者锁
一直在等待
被阻塞

```
      KERNEL: /mnt/vmlinux
    DUMPFILE: dump.201906230948  [PARTIAL DUMP]
        CPUS: 4
        DATE: Sun Jun 23 09:47:30 2019
      UPTIME: 01:19:34
LOAD AVERAGE: 2.41, 0.77, 0.30
       TASKS: 88
    NODENAME: benshushu
     RELEASE: 5.0.0-rlk+
     VERSION: #1 SMP Thu Jun 20 05:53:19 CST 2019
     MACHINE: aarch64  (unknown Mhz)
      MEMORY: 1 GB
       PANIC: "Kernel panic - not syncing: hung_task: blocked tasks"
         PID: 35
     COMMAND: "khungtaskd"
        TASK: ffff80002abf9d00  [THREAD_INFO: ffff80002abf9d00]
         CPU: 0
       STATE: TASK_RUNNING (PANIC)

crash>
```

发生死锁的原因是：有进程发生死锁，hung task and blocked
最后发生死锁的进程是：khungtaskd

kdump +crash 简单小结

1. **kdum + crash工具能帮忙我们解决死机黑屏（unresponsive）的问题**
2. **需要熟练掌握crash常用的命令**
3. **学会crash工具分析的基本流程**
4. **学会分析反汇编代码来找到函数调用参数在栈中存放的位置或者指针**
5. **放手去解决 死机黑屏问题吧！！**

实战9：企业级虚拟化计算节点服务器性能问题

实验目的：

➢通过一个企业级的例子来学习kdump在实际宕机问题解决的应用

➢本实战案例涉及到多个内核模块知识：kvm，NUMA-balance，KSM等

Enterprise Case Study

Migrating KSM page causes the VM lock up as the KSM page merging list is too large

https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1680513

Case Description

After **numad** is enabled and there are several VMs running on the same host machine, the **softlockup** messages can be observed inside the **VMs' dmesg**.

```
CPU: 3 PID: 22468 Comm: kworker/u32:2 Not tainted 4.4.0-47-generic #68-Ubuntu
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Ubuntu-1.8.2-1ubuntu1 04/01/2014
Workqueue: writeback wb_workfn (flush-252:0)
[<ffffffff81104388>] smp_call_function_many+0x1f8/0x260
[<ffffffff810727d5>] native_flush_tlb_others+0x65/0x150
[<ffffffff81072b35>] flush_tlb_page+0x55/0x90
```

Investigation on the VM side

This one seems a known issue. The bug is proactively handled by Linus when Dave Jones[3] issued the bug which happened on the bare metal machine. Tinoco[2] also found the bug in the nested KVM environment which happened when the IPI is sent out in the VCPU and it seems the problem coming from the LAPIC simulation of VMX. Chris Arges also involved in the debugging process and the debugging patch was given out by the Ingo Molnar, then Chris added some hacks to print out the debugging information. Unfortunately, after a long investigation, the root cause is still unknown.

[1]. smp/call: Detect stuck CSD locks https://patchwork.kernel.org/patch/6153801/

[2]. smp_call_function_single lockups https://lkml.org/lkml/2015/2/11/247

[3]. frequent lockups in 3.18rc4 https://lkml.org/lkml/2014/11/14/656

Investigation on the VM side

I've prepared a hotfix kernel which would resend the IPI and print out the information when the softlockup happens. **Unfortunately**, the hotfix kernel **doesn't** print out the **error message**. **That means my original thoughts are incorrect!**

The hotfix kernel source:

http://kernel.ubuntu.com/git/gavinguo/ubuntu-xenial.git/log/?h=sf000103690-csd-lock-debug

Host Machine - Hung task Backtrace

As I cannot find the clue inside the VMs, then try to investigate the host side.

Host Machine - Hung task Backtrace

```
# ksmd
crash> bt 615
PID: 615 TASK: ffff881fa174a940 CPU: 15 COMMAND: "ksmd"
#0 [ffff881fa1087cc0] __schedule at ffffffff818207ee
#1 [ffff881fa1087d10] schedule at ffffffff81820ee5
#2 [ffff881fa1087d28] rwsem_down_read_failed at ffffffff81823d60
#3 [ffff881fa1087d98] call_rwsem_down_read_failed at ffffffff813f8324
#4 [ffff881fa1087df8] ksm_scan_thread at ffffffff811e613d
#5 [ffff881fa1087ec8] kthread at ffffffff810a0528
#6 [ffff881fa1087f50] ret_from_fork at ffffffff8182538f
```

Host Machine - Hung task Backtrace

```
# khugepaged
crash> bt 616
PID: 616 TASK: ffff881fa1749b80 CPU: 11 COMMAND: "khugepaged"
#0 [ffff881fa108bc60] __schedule at ffffffff818207ee
#1 [ffff881fa108bcb0] schedule at ffffffff81820ee5
#2 [ffff881fa108bcc8] rwsem_down_write_failed at ffffffff81823b32
#3 [ffff881fa108bd50] call_rwsem_down_write_failed at ffffffff813f8353
#4 [ffff881fa108bda8] khugepaged at ffffffff811f58ef
#5 [ffff881fa108bec8] kthread at ffffffff810a0528
#6 [ffff881fa108bf50] ret_from_fork at ffffffff8182538f
```

Host Machine – Hung task Backtrace

```
# qemu-system-x86

crash> bt 12555
PID: 12555 TASK: ffff885fa1af6040 CPU: 55 COMMAND: "qemu-system-x86"
#0 [ffff885f9a043a50] __schedule at ffffffff818207ee
#1 [ffff885f9a043aa0] schedule at ffffffff81820ee5
#2 [ffff885f9a043ab8] rwsem_down_read_failed at ffffffff81823d60
#3 [ffff885f9a043b28] call_rwsem_down_read_failed at ffffffff813f8324
#4 [ffff885f9a043b88] kvm_host_page_size at ffffffffc02cfbae [kvm]
#5 [ffff885f9a043ba8] mapping_level at ffffffffc02ead1f [kvm]
#6 [ffff885f9a043bd8] tdp_page_fault at ffffffffc02f0b8a [kvm]
#7 [ffff885f9a043c50] kvm_mmu_page_fault at ffffffffc02ea794 [kvm]
#8 [ffff885f9a043c80] handle_ept_violation at ffffffffc01acda3 [kvm_intel]
#9 [ffff885f9a043cb8] vmx_handle_exit at ffffffffc01afdab [kvm_intel]
#10 [ffff885f9a043d48] vcpu_enter_guest at ffffffffc02e026d [kvm]
#11 [ffff885f9a043dc0] kvm_arch_vcpu_ioctl_run at ffffffffc02e698f [kvm]
#12 [ffff885f9a043e08] kvm_vcpu_ioctl at ffffffffc02ce09d [kvm]
#13 [ffff885f9a043ea0] do_vfs_ioctl at ffffffff81220bef
#14 [ffff885f9a043f10] sys_ioctl at ffffffff81220e59
```

Host Machine – Hung task Backtrace

We can see that the previous three tasks are waiting on the **mmap_sem**. The most interesting part is the backtrace of **numad**:

crash> bt 2950 The disassembly analysis of numad call stack

#1 [ffff885f8fb4fb78] smp_call_function_many
#2 [ffff885f8fb4fbc0] native_flush_tlb_others
#3 [ffff885f8fb4fc08] flush_tlb_page
#4 [ffff885f8fb4fc30] ptep_clear_flush
#5 [ffff885f8fb4fc60] try_to_unmap_one
#6 [ffff885f8fb4fcd0] rmap_walk_ksm
#7 [ffff885f8fb4fd28] rmap_walk
#8 [ffff885f8fb4fd80] try_to_unmap
#9 [ffff885f8fb4fdc8] migrate_pages

KSM merge list extraction

I've tried to **disassemble** the code and finally find the stable_node->hlist is as long as 2306920 entries(**Around 9.2GB memory merged into one page**).

rmap_item list(stable_node->hlist):

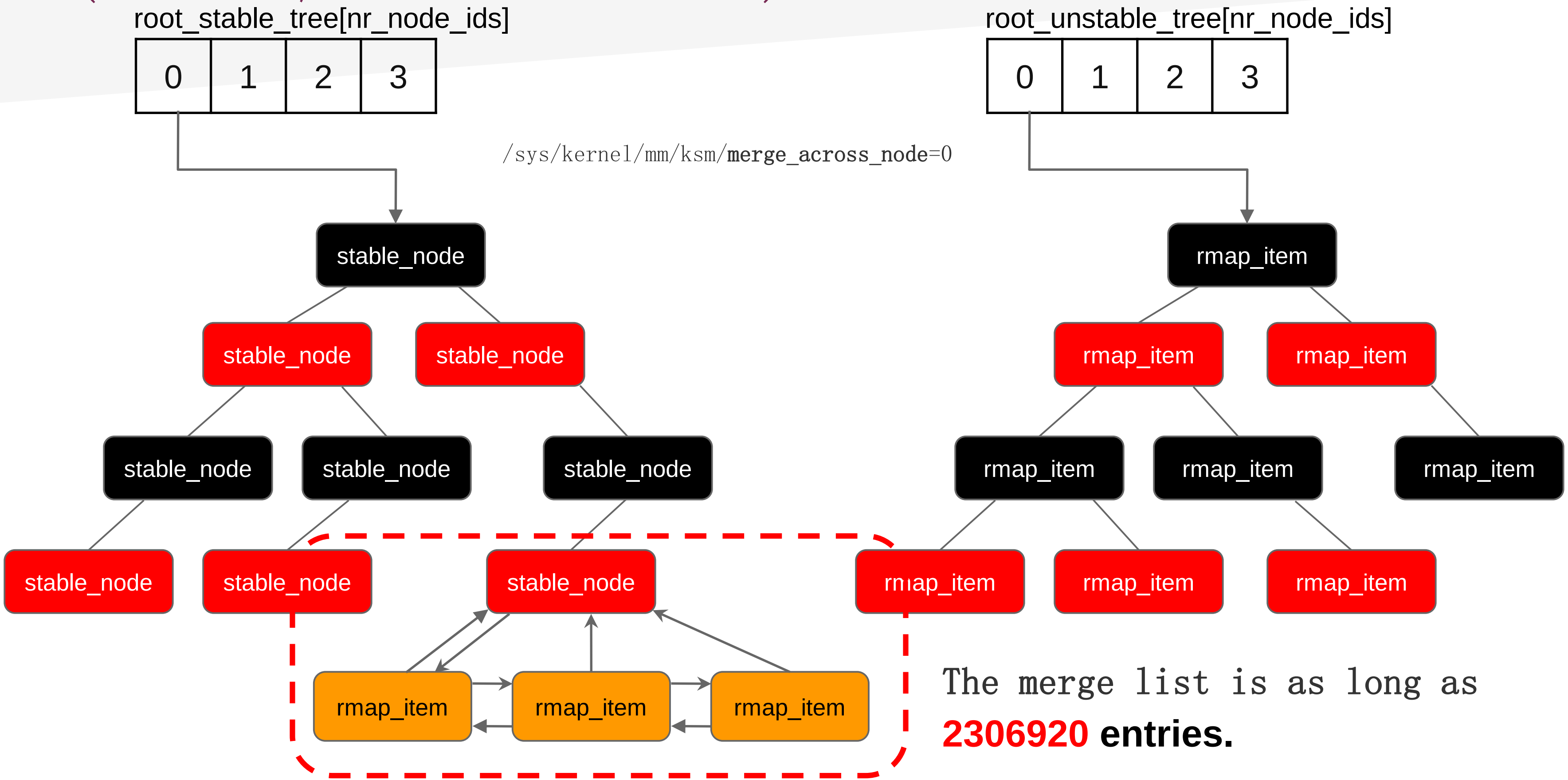
stable_node: 0xffff881f836ba000 stable_node->hlist->first = 0xffff883f3e5746b0

```
struct hlist_head {
    [0] struct hlist_node *first;
}
```

```
struct hlist_node {
[0] struct hlist_node *next;
[8] struct hlist_node **pprev;
}
```

```
crash> list hlist_node.next 0xffff883f3e5746b0 > rmap_item.lst
$ wc -l rmap_item.lst
2306920 rmap_item.lst
```

Introduction to the KSM Stable Tree (Stable/Unstable tree)

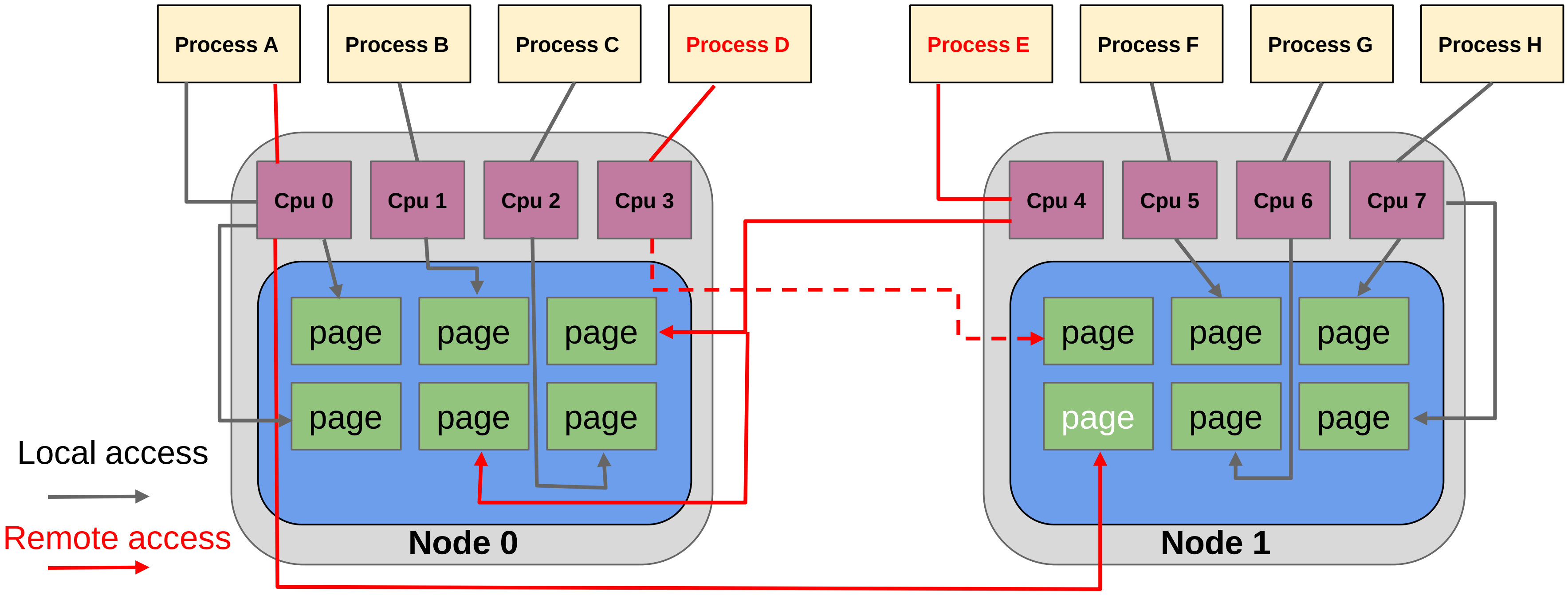


The merge list is as long as **2306920 entries.**

Automatic NUMA balancing

Local/Remote access

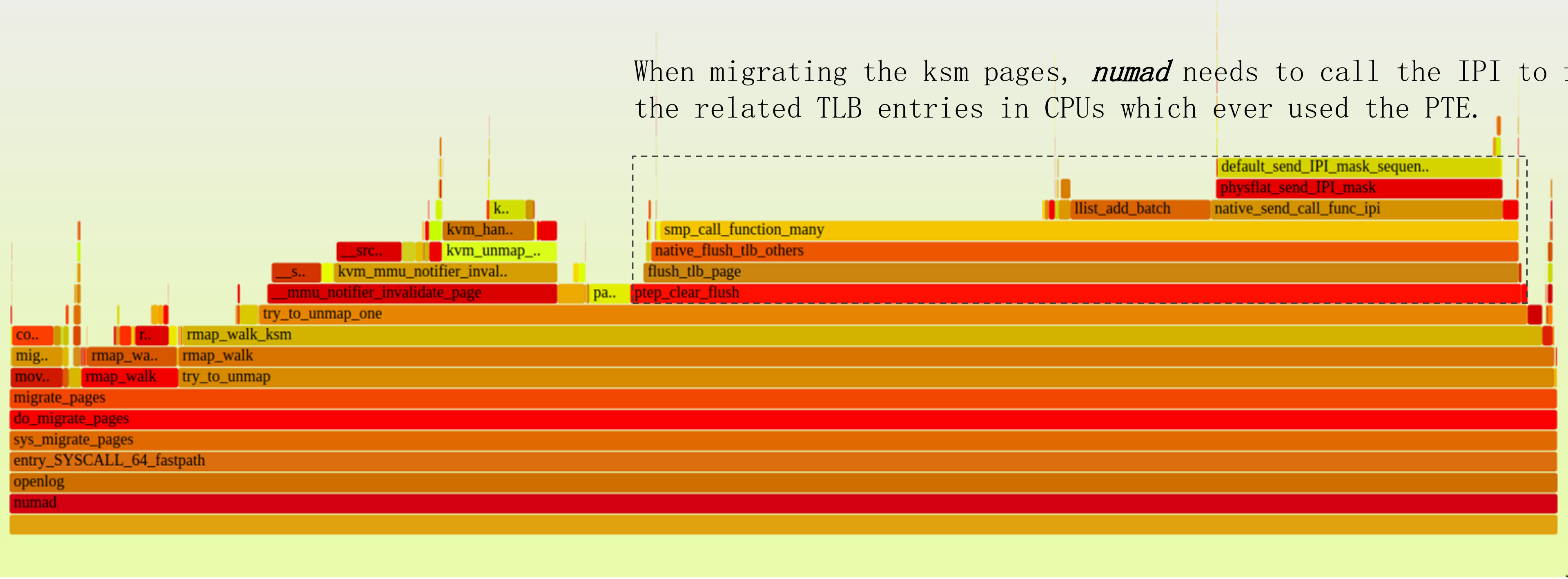
According to the memory access latency, it would be better to migrate Process D to node 1 and Process E to node 0. The remote access page by Process A can be migrated to node 0. However, it would also need to consider the CPU loading before migrating the processes.



FlameGraph of the performance problem

https://kernel.ubuntu.com/~gavinguo/sf00131845/numa-131845.svg

When migrating the ksm pages, *numad* needs to call the IPI to f
the related TLB entries in CPUs which ever used the PTE.



Solution

Re: [PATCH 1/1] ksm: introduce ksm_max_page_sharing per page deduplication limit
https://www.spinics.net/lists/linux-mm/msg125880.html

80b18dfa53bb ksm: optimize refile of stable_node_dup at the head of the chain

8dc5ffcd5a74 ksm: swap the two output parameters of chain/chain_prune

0ba1d0f7c41c ksm: cleanup stable_node chain collapse case

b4fecc67cc56 ksm: fix use after free with merge_across_nodes = 0

2c653d0ee2ae ksm: introduce ksm_max_page_sharing per page deduplication limit

Thank you

BACKUP

死机黑屏专题 视频课程

- 提供解决产品研发和服务器中遇到的死机问题的解决思路和方法
- 全程5小时高清视频
- 8大实验案例
- 140多页PPT
- 基于x86和ARM64
- 适用于Centos 和Ubuntu系统，以及ARM64的嵌入式系统

https://shop115683645.taobao.com/

微信号: Runing-LinuxKernel

更多更精彩更in的内容，请关注奔跑吧Linux社区微信公众号


奔跑吧
LINUX社区
旗舰篇一次订阅， 持续更新
微信号: Runing-LinuxKernel

背景知识：arm64体系结构的通用寄存器

提供31个64比特的通用寄存器

➢每个通用寄存器，用x表示64位宽，w表示32位宽

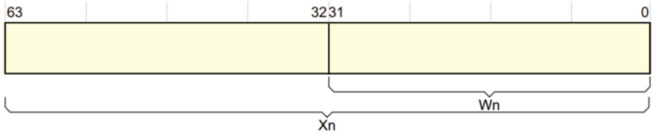


Figure 4-2 64-bit register with W and X access.

X0-X7	X8-X15	X16-X23	X24-X30
Parameter / result registers (X0-7) (Otherwise corruptible)	XR (X8)	IP0 (X16)	Callee-saved (X24-28)
	Corruptible Registers (X9-15)	IP1 (X17)	
		PR (X18)	
		Callee-saved (X19-23)	FP (X29) (callee-saved)
			LR (X30)

arm64函数参数调用规则

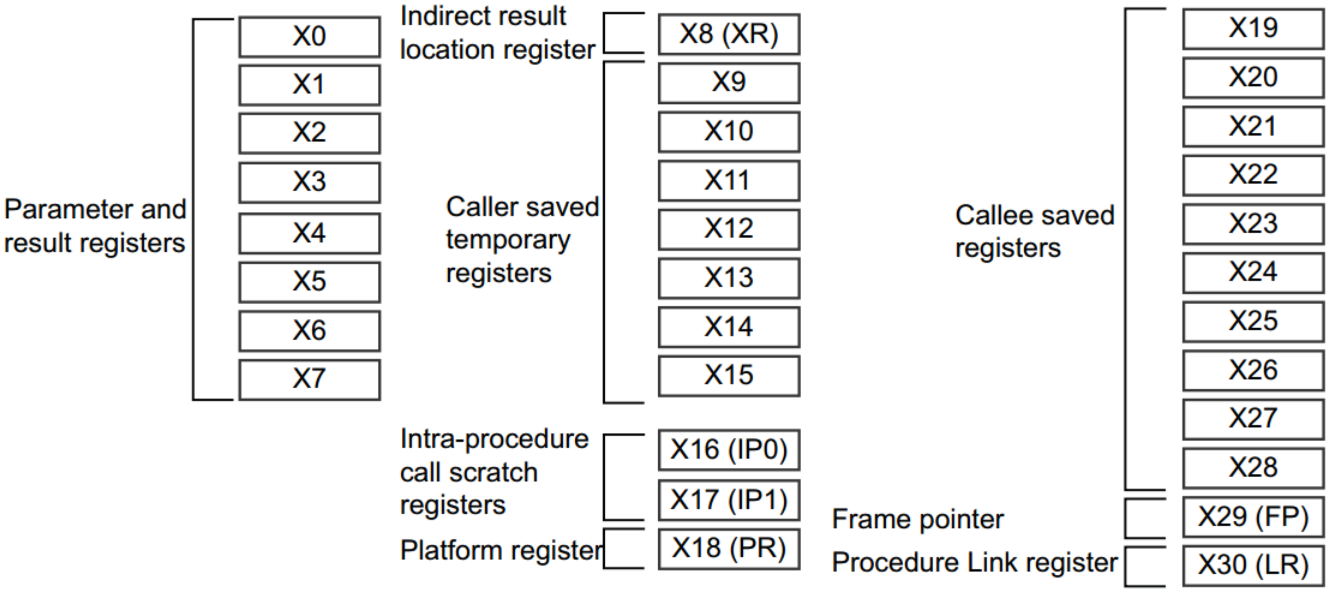


Figure 9-1 General-purpose register use in the ABI

ARM64通用寄存器

- ✓ **X0~X7：用于传递子程序参数和结果，使用时不需要保存，多余参数采用堆栈传递，64位返回结果采用X0表示，128位返回结果采用X1:X0表示。**
- ✓ X8：用于保存子程序返回地址， 尽量不要使用 。
- ✓ X9~X15：临时寄存器，使用时不需要保存。
- ✓ X16~X17：子程序内部调用寄存器，使用时不需要保存，尽量不要使用。
- ✓ X18：平台寄存器，它的使用与平台相关，尽量不要使用。
- ✓ X19~X28：临时寄存器，使用时必须保存。
- ✓ **X29：帧指针寄存器，用于连接栈帧，使用时需要保存。**
- ✓ **X30：链接寄存器LR**
- ✓ **X31：堆栈指针寄存器SP**