



www.yomocode.com

Linux常见锁和lockup检测机制

围绕死锁原理、**hung_task**、**softlockup/hardlockup** 的检测原理并
结合样例分享

JEFF XIE

主要内容

- Linux内核中的常见锁
- 死锁原理
- hungtask检测原理
- softlockup/hardlockup检测原理
- 死锁实验分享(使用qemu工具收集vmcore, crash工具分析)

Linux内核中常见锁 - 自旋锁

普通自旋锁 **spinlock** :

```
struct spinlock {  
    union {  
        struct raw_spinlock rlock;  
    };  
} spinlock_t;  
  
struct raw_spinlock {  
    arch_spinlock_t raw_lock;  
}
```

- 获取锁的过程（即上锁的过程）是自旋(忙等)的，不会引起睡眠和调度
- 持有自旋锁的临界区中不允许调度和睡眠，自旋锁的加锁操作会禁止抢占，解锁操作时再恢复抢占。自旋锁既可以用于进程上下文，又可以用于中断上下文
- 自旋锁的主要用途是多处理器之间的并发控制，适用于锁竞争不太激烈的场景。如果锁竞争非常激烈，那么大量的时间会浪费在加锁自旋上面，导致整体性能下降

Linux内核中常见锁 - 自旋锁

自旋锁spinlock API:

DEFINE_SPINLOCK(lock): 静态定义一个名为lock的自旋锁

spin_lock_init(lock): 自旋锁初始化 (设置为未锁状态)

spin_lock(lock): 加锁操作, 加锁成功后返回, 否则一直自旋(忙等)

spin_lock_irqsave(lock, flags) 加锁操作, 并关闭硬中断

spin_lock_bh(lock) 加锁操作, 并关闭软中断

spin_unlock(lock): 解锁操作, 解锁过程无竞争因此必然会成功

spin_unlock_irqrestore(lock, flags) 解锁操作, 并打开硬中断

spin_unlock_bh(lock) 解锁操作, 并打开软中断

读写自旋锁:rwlock_t

普通自旋锁的缺点:

- 对所有的竞争者不做区分。
- 很多情况下有些竞争者并不会修改共享资源（只读不写）
- 普通自旋锁总是会限制只有一个内核路径持有锁，而实际上这种限制是没有必要的

读写自旋锁的改进:

- 允许多个读者同时持有读锁（允许多个读者同时进入读临界区）
- 只允许一个写者同时持有写锁（只允许一个写者同时进入写临界区）
- 不允许读者和写者同时持有锁。
- 与普通自旋锁相比，读写自旋锁更适合读者多，写者少的应用场景。

读写自旋锁API:

- `DEFINE_RWLOCK(lock):`
静态定义一个名为lock的自旋锁。
- `rwlock_init(lock)`
自旋锁初始化（设置为未锁状态）。
- `read_lock(lock)`
加读锁操作，加锁成功后返回，否则一直自旋。
- `write_lock(lock)`
加写锁操作，加锁成功后返回，否则一直自旋。
- `read_unlock(lock)`
- `write_unlock(lock)` 解读锁操作，解锁过程无竞争因此必然会成功。
解写锁操作，解锁过程无竞争因此必然会成功。

Linux内核中常见锁 – 信号量

信号量

- Linux内核中应用最广泛的同步原语：自旋锁、信号量（semaphore）
- 自旋锁和信号量是一种互补的关系，它们有各自适用的场景
- 信号量可以是多值的，当其用作二值信号量时，类似于锁：一个值代表未锁，另一个值代表已锁
- 工作原理与自旋锁相反
 - ✓ 获取锁的过程中，若不能立即得到锁，就会发生调度，转入睡眠
 - ✓ 另外的内核执行路径释放锁时，唤醒等待该锁的执行路径
- 自旋锁的使用限制及信号量的解决方法
 - ✓ 持有自旋锁的临界区不允许调度和睡眠；竞争激烈时整体性能不好
 - ✓ 信号量解决了以上两个问题
 - 锁的竞争者不是忙等，信号量的临界区允许调度和睡眠而不会导致死锁
 - 锁的竞争者会转入睡眠，从而让出CPU资源给别的内核执行路径，因此对锁的竞争不会影响整体性能
- ✓ 信号量的缺点
 - 中断上下文要求整体运行时间可预测（不能太长），而信号量临界区可能发生调度，因此不能用于中断上下文（只能用于进程上下文）
 - 如果抢锁的过程很短，那么用信号量并不合算，因为进程睡眠加上唤醒的代价太大，消耗的CPU资源可能远远大于短时间的忙等

Linux内核中常见锁 – 信号量

普通信号量

```
struct semaphore {  
    raw_spinlock_t      lock;  
    unsigned int        count;  
    struct list_head    wait_list;  
};
```

- ❑ count标识了信号量的状态：值为0表示忙（已锁），值为正代表自由（未锁，允许竞争者进入临界区）。
- ❑ count的初值就是最大允许进入临界区的进程数目，初值为1的信号量就是二值信号量。
- ❑ 二值信号量类似于一个普通的锁，而多值信号量类似于一个允许一定并发性的锁。
- ❑ wait_list字段是当信号量为忙时，所有等待信号量的进程列表，而lock则是保护wait_list的自旋锁。

Linux内核中常见锁 – 信号量

普通信号量的主要API:

- `DEFINE_SEMAPHORE(sem)`: 静态定义一个名为sem信号量。
- `void sema_init(struct semaphore *sem, int val)`: 初始化一个信号量sem, 计数器初值为val。
- `void down(struct semaphore *sem)`: 减少信号量sem的计数器 (类似于获取锁)。如果失败 (计数器已经是0) , 那么转入睡眠 (状态为`TASK_UNINTERRUPTIBLE`, 不会被任何信号唤醒) 并把当其进程挂到`wait_list`; 被唤醒后继续尝试获取锁。
- `void up(struct semaphore *sem)`: 增加信号量sem的计数器 (类似于释放锁) , 然后唤醒`wait_list`里面的第一个进程 (如果有的话) 。

Linux内核中常见锁 – 信号量

读写信号量

读写信号量的引入原因类似于读写自旋锁，是为了区分不同的竞争者（读者和写者），以便允许读者共享而写者互斥。

```
struct rw_semaphore {  
    long count;  
    struct list_head wait_list;  
    raw_spinlock_t wait_lock;  
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER  
    struct optimistic_spin_queue osq;  
    struct task_struct *owner;  
#endif  
};
```

主要字段count、wait_list和wait_lock的含义与普通信号量基本相同,owner指向写者进程。

Linux内核中常见锁 – 信号量

读写信号量的主要**API**:

- ❑ `DECLARE_RWSEM(sem)`: 静态声明一个名为sem信号量。
- ❑ `init_rwsem(sem)`: 初始化一个信号量sem。
- ❑ `down_read(struct rw_semaphore *sem)`: 读者减少信号量sem的计数器 (类似于获取锁)
- ❑ `down_write(struct rw_semaphore *sem)`: 写者减少信号量sem的计数器 (类似于获取锁)
- ❑ `up_read(struct rw_semaphore *sem)`: 读者增加信号量sem的计数器 (类似于释放锁)
- ❑ `up_write(struct rw_semaphore *sem)`: 写者增加信号量sem的计数器 (类似于释放锁)

Linux内核中常见锁 – 信号量

互斥量

互斥量是二值信号量

```
struct mutex {  
    atomic_t      count;  
    spinlock_t    wait_lock;  
    struct list_head    wait_list;  
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER  
    struct task_struct  *owner;  
    struct optimistic_spin_queue osq;  
#endif  
};
```

在数据结构上与信号量几乎相同。

Linux内核中常见锁 – 信号量

互斥量API

互斥量提供了一套新的API，这套API专为二值的互斥量优化。这些API主要有：

- ❑ `DEFINE_MUTEX(mutex):` 静态定义一个名为mutex的互斥量。
- ❑ `mutex_init(mutex):` 初始化一个互斥量mutex，初始状态为未锁。
- ❑ `void mutex_lock(struct mutex *lock):` 对互斥量加锁，如果失败，那么转入睡眠（状态为`TASK_UNINTERRUPTIBLE`，不会被任何信号唤醒）并把当其进程挂到`wait_list`。
- ❑ `void mutex_unlock(struct mutex *lock):` 对互斥量解锁，然后唤醒`wait_list`里面的第一个进程（如果有的话）。

死锁原理

典型例子：

- spinlock rwlock mutex 在linux内核中是不能递归的,如果产生递归会发生死锁

例如调用spinlock(A),在本cpu上发生中断，中断上下文也调用到spinlock(A)

- | CPU 1 | CPU2 |
|------------------|------------------|
| 获取lock A -> OK | 获取lock B -> OK |
| 获取lock B -> spin | 获取lock A -> spin |

hungtask检测原理

```
#ps -o pid,comm,cls,ni -C khungtaskd
PID COMMAND      CLS NI
 38 khungtaskd    TS  0
```

```
#cd /proc/sys/kernel/
#grep . hung*
hung_task_check_count:4194304 (检查的次数)
hung_task_check_interval_secs:0 (每隔多长时间检查一次)
hung_task_panic:0 (设置进程hung之后是否触发panic)
hung_task_timeout_secs:0 (进程hung的超时时间)
hung_task_warnings:10 (警告的次数)
```

```
/etc/sysctl.conf
kernel.hung_task_panic = 1
Kernel.hung_task_timeout_secs = 120
```

使用进程 khungtaskd内核配置选项:CONFIG_DETECT_HUNG_TASK which are bugs that cause the task to be stuck in uninterruptible "D" state indefinitely.

```
kernel/hung_task.c
watchdog()
set_user_nice(current, 0);
for(;;) {
    check_hung_uninterruptible_tasks(timeout)
    for_each_process_thread(g, t) {
        if (t->state == TASK_UNINTERRUPTIBLE)
            是否超过hung_task_timeout_secs
            打印D状态堆栈,或者同时panic
```

```
#ps -eLf r 查看r状态和D状态的进程  
#cat /proc/pid/stack 查看进程内核堆栈  
  
#ps -eLf r  
root 4394 4393 D+ 0:00 insmod ./hungtask.ko
```

hungtask样例

```
○[ 2055.258547] INFO: task insmod:4394 blocked for more than 120  
seconds.  
○[ 2055.258552]  Tainted: G      OE  N 5.3.18-24.34-default #1  
○[ 2055.258553] "echo 0 > /proc/sys/kernel/hung_task_timeout_secs"  
disables this message.  
○[ 2055.258555] task:insmod      state:D stack: 0 pid: 4394 ppid:  
4393 flags:0x80004004  
○[ 2055.258557] Call Trace:  
○[ 2055.258563] __schedule+0x2fd/0x750  
○[ 2055.258567] ? 0xfffffffffc0b3b000  
○[ 2055.258569] schedule+0x2f/0xa0  
○[ 2055.258571] hungtask_init+0xa/0x1000 [hungtask]  
○[ 2055.258575] do_one_initcall+0x46/0x1f4  
○[ 2055.258578] ? kmem_cache_alloc_trace+0x43/0x260  
○[ 2055.258581] ? do_init_module+0x22/0x21f  
○[ 2055.258584] do_init_module+0x5b/0x21f  
○[ 2055.258586] load_module+0x1d6a/0x2320  
○[ 2055.258590] ? ima_post_read_file+0xe2/0x120  
○[ 2055.258592] ? __do_sys_finit_module+0xe9/0x110  
○[ 2055.258594] __do_sys_finit_module+0xe9/0x110  
○[ 2055.258597] do_syscall_64+0x5b/0x1e0  
○[ 2055.258599] entry_SYSCALL_64_after_hwframe+0x44/0xa9
```

```
#cat /proc/4394/stack  
hungtask_init+0xa/0x1000 [hungtask]  
do_one_initcall+0x46/0x1f4  
do_init_module+0x5b/0x21f  
load_module+0x1d6a/0x2320  
__do_sys_finit_module+0xe9/0x110  
do_syscall_64+0x5b/0x1e0  
entry_SYSCALL_64_after_hwframe+0x44/0xa9
```

```
#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/sched.h>  
  
static noinline void hungtask(void)  
{ set_current_state(TASK_UNINTERRUPTIBLE);  
    schedule();  
}  
static int __init hungtask_init(void)  
{  
    hungtask();  
    return 0;  
}  
static void __exit hungtask_exit(void)  
{  
    pr_info("%s\n", __func__);  
}  
module_init(hungtask_init);  
module_exit(hungtask_exit);  
MODULE_AUTHOR("Jeff Xie");  
MODULE_LICENSE("GPL");
```

什么是softlockup和hardlockup ?

softlockup(软死锁):

softlockups are bugs that cause the kernel to loop in kernel mode for more than 20 seconds, without giving other tasks a chance to run. The current stack trace is displayed upon detection and the system will stay locked up.

例如持有spinlock之后在临界区花了太长时间

hardlockup(硬死锁):

hardlockups are bugs that cause the CPU to loop in kernel mode for more than 10 seconds, without letting other interrupts have a chance to run. The current stack trace is displayed upon detection and the system will stay locked up.

例如关闭本地中断太长时间

softlockup/hardlockup检测原理

hrtimer: watchdog_timer_fn() (每cpu上都有一个)

softlockup: softlockup_fn()

hardlockup: watchdog_overflow_callback()

hrtimer (watchdog_timer_fn)

- 默认每4s触发一次
- (唤醒migration/N 进程,stop调度类进程)
- 对hardlockup需要检查的hrtimer_interrupts计数器进行加一.

```
set_sample_period() {  
get_softlockup_thresh() *  
((u64)NSEC_PER_SEC / 5);(4s)}
```

softlockup检查机制:

(CONFIG_SOFTLOCKUP_DETECTOR)

- Migration/N 重置watchdog_touch_ts 时间戳
- hrtimer执行路径检查到20s内 watchdog_touch_ts时间戳没有更新,说明 migration/N在20s之内都没有被唤醒执行
- hrtimer执行路径打印softlockup的cpu堆栈或者同时panic.

```
get_softlockup_thresh() {watchdog_thresh * 2 }  
(20s),
```

```
#grep . /proc/sys/kernel/soft*  
/proc/sys/kernel/soft_watchdog:1  
/proc/sys/kernel/softlockup_all_cpu_backtrace  
:0  
/proc/sys/kernel/softlockup_panic:0
```

Hardlockup检查机制:

(CONFIG_HARDLOCKUP_DETECTOR)

- 默认每10s (watchdog_thresh)检查系统是否存在 hardlockup
- 如果检查到hrtimer没有执行(有2.5次执行机会,执行时会对计数器加一 __this_cpu_inc(hrtimer_interrupts);) 则会打印hardlockup的cpu堆栈信息或者同时 panic.

```
#cat /proc/sys/kernel/watchdog_thresh  
10
```

```
#grep . /proc/sys/kernel/hardlockup_*  
/proc/sys/kernel/hardlockup_all_cpu_backtrace:0  
/proc/sys/kernel/hardlockup_panic:1
```

softlockup检查机制 (migration/N 进程 stop调度类)

```
schedule()
pick_next_task()
for_each_class(class) {
    p = class->pick_next_task(rq);
    if (p)
        return p;
}
#define sched_class_highest __stop_sched_class
#define sched_class_lowest __idle_sched_class
#define for_each_class(class) \
    for_class_range(class, sched_class_highest, sched_class_lowest)
void sched_set_stop_task(int cpu, struct task_struct *stop)
    * Make it appear like a SCHED_FIFO task, its something
    * userspace knows about and won't get confused about.
sched_setscheduler_nocheck(stop, SCHED_FIFO, &param);
stop->sched_class = &stop_sched_class;
```

```
#uname -r
3.10.0-1062.4.1.el7.x86_64
#ps -ef | grep watchdog/
root      11  2 0 May28  00:01:45 [watchdog/0]
root      12  2 0 May28  00:01:13 [watchdog/1]
root      17  2 0 May28  00:01:08 [watchdog/2]
root      22  2 0 May28  00:01:06 [watchdog/3]
#ps -e -o pid,comm,cls | grep migration
12 migration/0  FF
17 migration/1  FF
23 migration/2  FF
29 migration/3  FF
35 migration/4  FF
...
```

Hardlockup检测使用perf子系统

```
hardlockup_detector_perf_enable
hardlockup_detector_event_create
    wd_attr->sample_period = hw_nmi_get_sample_period(watchdog_thresh);
    perf_event_create_kernel_counter(wd_attr, cpu, NULL,
        watchdog_overflow_callback, NULL);

watchdog_update_hrtimer_threshold(sample_period);
/* The NMI watchdog on x86 is based on unhalted CPU cycles */

#cat /proc/interrupts | grep NMI
NMI:    1      1      1      2  Non-maskable interrupts
```

在X86平台上，缺省的空闲进程就是执行HLT指令（相当于进入C1）而暂停CPU运行，但如果空闲持续的时间较长，则可以选择更深的C-State。C0才是在unhalted core cycle. 所以在系统空闲时，不会触发NMI，进而hardlockup检查机制使用的nmi中断在系统idle时不会默认10s触发控制pmu(performance monitoring unit)的nmi section 6.7

<https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html>

softlockup/hardlockup检测代码逻辑

```
watchdog_timer_fn()  
  
unsigned long touch_ts =  
    __this_cpu_read(watchdog_touch_ts);  
  
/* kick the hardlockup detector */  
    watchdog_interrupt_count();  
    __this_cpu_inc(hrtimer_interrupts);  
  
/* kick the softlockup detector */  
stop_one_cpu_nowait(smp_processor_id(),  
    softlockup_fn, NULL,  
  
this_cpu_ptr(&softlockup_stop_work));  
  
is_softlockup(touch_ts);  
    dump_stack();  
if (softlockup_panic)  
    panic("softlockup: hung tasks");
```

```
softlockup_fn()  
    __this_cpu_write(watchdog_touch_ts, get_timestamp());  
  
static int is_softlockup(unsigned long touch_ts)  
{  
    unsigned long now = get_timestamp();  
    if (time_after(now, touch_ts + get_softlockup_thresh()))  
        return now - touch_ts;  
    return 0;  
}  
  
bool is_hardlockup(void)  
{  
    unsigned long hrint = __this_cpu_read(hrtimer_interrupts);  
    if (__this_cpu_read(hrtimer_interrupts_saved) == hrint)  
        return true;  
    __this_cpu_write(hrtimer_interrupts_saved, hrint);  
    return false;  
}
```

=> watchdog_timer_fn
=> __hrtimer_run_queues
=> hrtimer_interrupt
=> smp_apic_timer_interrupt
=> apic_timer_interrupt
=> cpuidle_enter_state
=> cpuidle_enter
=> do_idle
=> cpu_startup_entry
=> start_kernel
=> secondary_startup_64

migrate/0 (stop_class)
=> softlockup_fn
=>
cpu_stopper_thread
=>
smpboot_thread_fn
=> kthread
=> ret_from_fork

(NMI_per event
=> watchdog_overflow_callback
=> __perf_event_overflow
=> handle_pmi_common
=> intel_pmu_handle_irq
=> perf_event_nmi_handler
=> nmi_handle
=> default_do_nmi
=> exc_nmi
=> asm_exc_nmi

softlockup/hardlockup 样例

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/interrupt.h>

static __attribute__((noinline)) void hardlockup_loop(void)
{
    pr_info("%s\n", __func__);
    for(;;);
}

static int __init hardlockup_init(void)
{
    local_irq_disable(); /* 去掉就是softlockup */
    hardlockup_loop();
    return 0;
}

static void __exit hardlockup_exit(void)
{
    pr_info("%s\n", __func__);
}

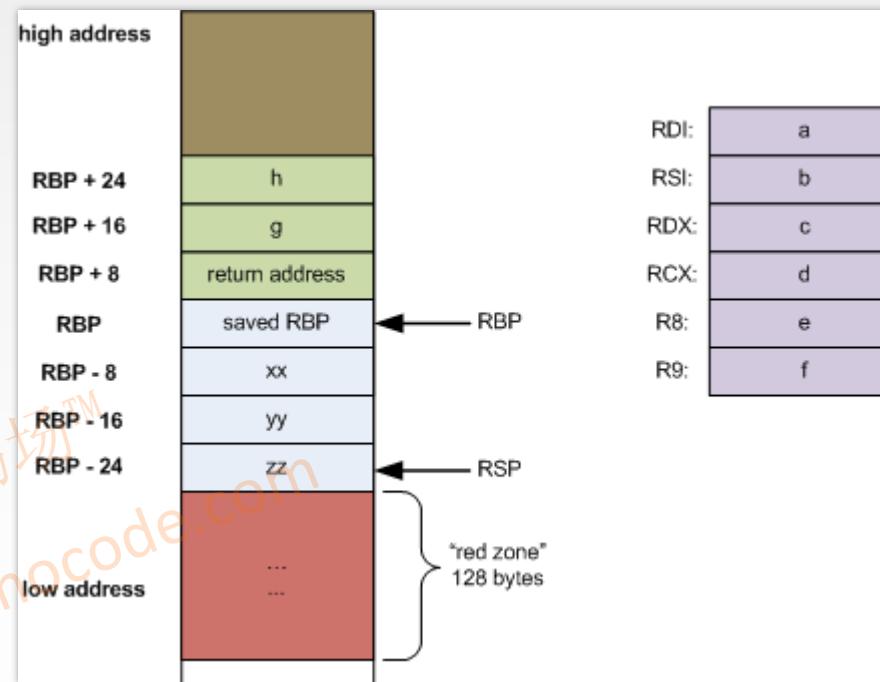
module_init(hardlockup_init);
module_exit(hardlockup_exit);
MODULE_AUTHOR("Jeff Xie");
MODULE_LICENSE("GPL");
```

死锁实验分享(使用qemu工具收集 vmcore, crash工具分析)

crash分析背景知识: (x86_64栈帧布局)

```
long myfunc(long a, long b, long c)
{
    long xx = a * b * c ;
    long yy = a + b + c;
    long zz = utilfunc(xx, yy, xx % yy);
    return zz + 20;
}

int main(void)
{
    long r;
    r = myfunc(1, 2, 3);
    ...
    return 0;
}
```



实验环境构建

crash版本: 7.2.9 (当前最高版本解析vmcore更好)

qemu版本: 5.1.50 (commit:b150cb8f67)

Linux kernel 基于linux5.5 (commit: d5226fa6dbae Linux 5.5)

#git patch -check ./0001-test-simulate-a-deadlock-case.patch (检查patch是否能打上)

#git am ./0001-test-simulate-a-deadlock-case.patch (打上patch)

#cp arch/x86/configs/x86_64_defconfig ./config (使用x86_64默认配置)

#ls ./kernel/configs

kvm_guest.config

#make kvm_guest.config (合并kvm客户端配置选项)

#make bzImage -j4 (编译内核)

./lockup.sh

```
#!/bin/bash
/home/jeff/nvme/git/qemu/build/x86_64-softmmu/qemu-system-x86_64 \
-qmp tcp:localhost:4444,server,nowait \
-cpu kvm64 \
-smp cores=4,threads=2 \
-m 256M \
-kernel ./linux-test/arch/x86/boot/bzImage \
-drive file=./kvm.img,if=virtio,format=raw \
-serial mon:stdio \
-vga std \
-append "root=/dev/vda rw iowait init=/linuxrc loglevel=8 console=ttyS0"
```

内核打开编译选项:

CONFIG_DEBUG_INFO

CONFIG_SOFTLOCKUP_DETECTOR

CONFIG_BOOTPARAM_SOFTLOCKUP_PANIC

使用qemu收集vmcore

收集vmcore到指定文件: /home/jeff/nvme/vmcore.img

```
#telnet localhost 4444
```

```
Trying ::1...
```

```
Connected to localhost.
```

```
Escape character is '^]'.
```

```
{"QMP": {"version": {"qemu": {"micro": 50, "minor": 1, "major": 5}, "package": ""}, "capabilities": ["oob"]}}
```

```
{"execute": "qmp_capabilities"}
```

```
{"return": {}}
```

```
{"execute": "dump-guest-
```

```
memory", "arguments": {"paging": false, "protocol": "file:/home/jeff/nvme/vmcore.img"}}
```

```
..."DUMP_COMPLETED"...
```

```
{"return": {}}
```

```
^]
```

```
telnet> quit
```

分析vmcore

```
#crash ./vmcore.img ./vmlinux
```

作业:

1. 用vim工具打开任意一个文件，使用crash在线调试，查看vim进程打开的文件内容
2. 使用qemu工具产生案例中的vmcore，并使用crash分析

死锁模拟patch:



0001-test-simulate-a-deadlock-case.patch

参考:

1. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/>
2. 《用芯探核》
3. <https://wiki.ubuntu.com/DebuggingKernelWithQEMU>
4. <https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64>



阅码场出品

www.yomocode.com