Kernel Mode Linux (KML)

施鹏飞

2023 年 3 月 22 日

目 录

01 KML 技术简介

- Kernel Mode Linux （以下简称 KML ）是一种允许在内核模式下运行用户程序的技术。
  - 使用 KML 技术的好处在于用户程序以特权级别执行，则可以直接访问内核地址空间，因此用户程序可以快速地直接使用系统调用，而无需花费代价高昂的软中断以及上下文切换在内核态和用户态之间切换。
  - KML 技术允许用户直接访问内核看似存在一些安全隐患，但是可以通过静态类型检查、软件故障隔离等方法来确保内核的安全性。 KML 技术的作者为此开发了一套基于 KML 以及 TAL(Typed assembly language，有类型的汇编语言 ）的安全系统，不过这点不在我们的讨论范围内。

01 KML 技术简介

- 在 KML 技术下， KML 用户程序除了拥有最高的特权优先级以外，其余地方都将仍然被当作一个一般的程序执行。
  - KML 用户程序拥有自己的地址空间，分页功能正常执行。
  - 异常（段错误、非法指令等）会被正常处理，比如：

```
int main(int argc, char* argv[])
{
    *(int*)0 = 1;
    return 0;
}
```

该进程会被正常杀死并抛出一个段错误异常，而不会导致 kernel pacnic

01 KML 技术简介

- 信号机制可以正常对 KML 程序工作，如：

```
int main(int argc, char* argv[])
{
    for (;;);
    return 0;
}
```

通过 <Ctrl-C> 发送 SIGINT 信号可以终止该程序执行，该例子也说明了进程调度系统也对 KML 程序正常工作，内核不会因此而卡死。

- 不要修改 CS 、 DS 、 SS 、 FS 等寄存器的内容，这些寄存器被默认认为不会被 KML 程序所修改。
- 不要胡乱执行特权指令（如只执行一次 cli 汇编指令而不执行 sti 等），不然系统可能卡死。

02 QEMU 与内核编译

- 本节我们将介绍：
  - 实验环境
  - 从源码编译内核
  - 制作 initrd 根文件系统
  - 使用 QEMU 调试内核

02 QEMU 与内核编译 - 实验环境

- **现成的发行版**

02 QEMU 与内核编译 - 实验环境

- 宿主机环境： x86_64 架构， ArchLinux
- QEMU emulator version 7.2.0
- Linux-3.18.140
  - Real-Time-3.18.140-rt177 补丁
  - KML-3.18 补丁
  - Gcc-4.4
- Busybox-1.33.2
  - Musl-gcc 12.2.1

- 此为一种参考环境，关于环境以及各类版本的尝试将在后续继续介绍。

02 QEMU 与内核编译 - 从源码编译内核

- **下载 Linux 源码：**

```
git clone -b v4.4.12 --depth 1 git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git
linux-4.4.12
```

- **下载 RT 补丁：**

```
wget https://cdn.kernel.org/pub/linux/kernel/projects/rt/4.4/older/patch-4.4.12-
rt18.patch.xz
```

- **打补丁：**

```
cd linux-4.4.12
xzcat ../patch-4.4.12-rt18.patch.xz | patch -p1
```

- **下载并打上 KML 补丁，操作与 RT 类似；**

02 QEMU 与内核编译 – 从源码编译内核

- **补充说明：**
  - PREEMPT_RT 内核补丁，目的是为了将 Linux 内核拓展为一个实时的操作系统。
    - 常规意义上来讲，Linux 并不算是一个实时操作系统，它并没有很强的实时处理能力，但是如果加入了 PREEMTP_RT 补丁后，它可以成为一个实时操作系统。 PREEMTP_RT 补丁并不属于内核主线，不同内核版本都有对应的 PREEMTP_RT 补丁，它是由单独项目维护的 (Real-Time Linux Project)。
    - 通过打上 PREEMTP_RT 补丁，然后重新配置内核的实时抢占模型，重新编译后才能成为一个实时系统。经过打上补丁， PREEMTP_RT Linux 是可以达到硬实时需求的。
  - 如果使用 sonicyang/KML 的补丁版本 (针对 Linux-4.4.12) ，据说必须要打上 RT 内核补丁才可以正常运行 KML 机制 .

02 QEMU 与内核编译 – 从源码编译内核

- **配置默认内核选项：**

```
export ARCH=x86_64;  make defconfig
```

  - ARCH 环境变量，用于决定编译出的内核使用的架构
  - make defconfig 做了什么？当 .config 文件被生成，内核构建系统仔细检查所有的 Kconfig 文件（从所有的子文件夹中），检查那些 Kconfig 中的所有选项：
    - 如果选项在 defconfig 中提及，构建系统将输出该选项和从 defconfig 从取出的该选项的值到 .config 文件中
    - 如果该选项没在 defconfig 中提及，构建系统使用它在 Kconfig 中对应的默认值到 .config 文件中。

02 QEMU 与内核编译 - 从源码编译内核

- **指定想要的内核选项，启用内核 debug， 关闭地址随机化：**

```
Kernel hacking  --->
    [*] Kernel debugging
    Compile-time checks and compiler options  --->
        [*] Compile the kernel with debug info
        [*]   Provide GDB scripts for kernel
debugging
Processor type and features ---->
    [] Randomize the address of the kernel image
(KASLR)
```

- **全速编译内核：**

```
make -j `nproc`
```

02 QEMU 与内核编译 - 从源码编译内核

Troubleshooting：不支持 PIC 模式

```
cc1: error: code model kernel does not support PIC mode
```

- 在 Makefile 中添加 gcc 编译参数 -fno-pie，其原因在于新版的 gcc（确切地说是 gcc6 及以后，尽管 gcc5 好像也有这个 bug）默认会使用 pie 技术，但是老版本的内核并不支持，因而必须在 Makefile 中手动关闭

```
KBUILD_CFLAGS   := -Wall -Wundef -Wstrict-
prototypes -Wno-trigraphs \
          -fno-strict-aliasing -fno-common \
          -Werror-implicit-function-declaration \
          -Wno-format-security \
          -std=gnu89 -fno-pie
```

02 QEMU 与内核编译 - 从源码编译内核

- Troubleshooting：不支持 PIC 模式
  - 查阅 GCC 官方文档学习一下 -fpic，-fPIC，-fpie，-fPIE 这四个选项的意思：
  - -fpic 会产生 Position-Independent Code(PIC)，通常用于共享库中。在该选项下，常量通过一个全局偏移表(global offset table，GOT)来访问，受到机器的限制，当该偏移表过大时，可以使用 -fPIC 选项，该选项允许更大的偏移表
  - -fpie，-fPIE 是类似的功能，不过该选项只会产生用于链接为可执行文件的代码。

02 QEMU 与内核编译 – 从源码编译内核

- 编译完成：
  - 修改完 PIC 后， 可以编译成功
  - 得到一个内核镜像， 其位置在 arch/x86_64/boot/bzImage

- 此时，仅仅拥有一个内核镜像还不足以使用 QEMU 启动整个系统，其还需要 initrd 根目录系统，这将在下一小节进行制作。
- 但在此之前，可以尝试去 QEMU 装载该内核看看如何是否可以成功解压、引导内核。

02 QEMU 与内核编译 - 从源码编译内核

- QEMU 装载 Linux 内核：

```
qemu-system-x86_64 -kernel ./bzImage
```

- 卡死，一直闪烁而无法进入内核

```
Booting from ROM...
  Probing EDD (edd=off to disable)... ok
  early console in extract_kernel
  input_data: 0x0000000002c773b4
  input_len: 0x000000000090c470
  output: 0x0000000001000000
  output_len: 0x000000000173a968
  kernel_total_size: 0x00000000025a7000

  Decompressing Linux... Parsing ELF...
```

02 QEMU 与内核编译 - 从源码编译内核

- Troubleshooting：无法解压内核
  - 造成这种问题的原因在于：
    - binutils 2.31 改变了链接器 (ld) 的默认值，例如 max-page-size 的默认值从 2MiB 降低到 4kiB，这个更改会破坏 x86_64 内核
  - 需要在 arch/x86/Makefile 中打上补丁：

```
+ifdef CONFIG_X86_64
+LDFLAGS += $(call ld-option, -z max-page-size=0x200000)
+endif
```

  - 重新编译内核，QEMU 可以成功装载！

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- 为什么需要 initrd(initrd ramdisk) 根文件系统?
  - Linux 启动阶段，boot loader 加载完内核文件 vmlinuz 之后，便开始挂载磁盘根文件系统。挂载操作需要磁盘驱动，所以挂载前要先加载驱动。但是驱动位于 /lib/modules，不挂载磁盘就访问不到，形成了一个死循环。
  - initramfs 根文件系统就可以解决这个问题，其中包含必要的设备驱动和工具，boot loader 会加载 initramfs 到内存中，内核将其挂载到根目录，然后运行 /init 初始化脚本，去挂载真正的磁盘根文件系统。

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Hello, Kernel!
  - 引入 Busybox 工具包建立一个完整的文件系统较为复杂，在此之前，我们可以制作一个 HelloWord 文件系统，并以此说明一些 QEMU 的参数选项

```
$ gcc -static -o hello hello.c
$ echo hello | cpio -o --format=newc > rootfs
$ qemu-system-x86_64 -kernel ./bzImage -initrd
./rootfs \
    -append "root=/dev/ram rdinit=/hello"
```

  - qemu 的 -kernel 和 -initrd 能够绕过 bootload 直接对指定的 kernel 和 ramdisk 进行加载.
  - 用 -append 进行额外的选项配置，在这里我们把根目录直接设置成 /tmp 目录，启动的 init 程序设置成放进去的 hello 程序

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Hello, Kernel!
  - 成功!
  - 但没什么用!

```
Gtk-Message: 20:19:15.584: Failed to load module "window-decorations-gtk-module"
^Cqemu: terminating on signal 2
rqdmap in ~/KML via v12.2.1-gcc took 10s
[ 20:19:25 ] qemu-system-x86_64 -kernel ./bzImage -initrd ./rootfs -append "root=/tmp rdinit=/hello"
Gtk-Message: 20:19:26.018: Failed to load module "appmenu-gtk-module"
Gtk-Message: 20:19:26.040: Failed to load module "colorreload-gtk-module"
Gtk-Message: 20:19:26.040: Failed to load module "window-decorations-gtk-module"
```

Machine View

```
[    0.795440] Key type dns_resolver registered
[    0.796387] microcode: AMD CPU family 0xf not supported
[    0.797446] registered taskstats version 1
[    0.801218]   Magic number: 15:972:328
[    0.801487] console [netcon0] enabled
[    0.801528] netconsole: network logging started
[    0.802585] ALSA device list:
[    0.802619]   No soundcards found.
[    0.850245] Freeing unused kernel memory: 1068K (ffffffff81f1f000 - ffffffff8
202a000)
[    0.850317] Write protecting the kernel read-only data: 14336k
[    0.851273] Freeing unused kernel memory: 1428K (ffff88000189b000 - ffff88000
1a00000)
[    0.856568] Freeing unused kernel memory: 1324K (ffff880001cb5000 - ffff88000
1e00000)
Hello Kernel!
Hello Kernel!
Hello Kernel!
[    1.328271] tsc: Refined TSC clocksource calibration: 2687.991 MHz
[    1.328335] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x26beee0a
6f1, max_idle_ns: 440795337953 ns
[    1.373635] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[    2.332362] clocksource: Switched to clocksource tsc
```

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Busybox 简介
  - BusyBox 是一个遵循 GPL 协议、以自由软件形式发行的应用程序。 Busybox 在单一的可执行文件中提供了精简的 Unix 工具集，可运行于多款 POSIX 环境的操作系统。由于 BusyBox 可执行文件的文件比较小，使得它非常适合使用于嵌入式系统。作者将 BusyBox 称为“嵌入式 Linux 的瑞士军刀”
  - BusyBox 是单一二进制程序，它是许多应用程序的集合，在适当参数的特定方法下，每个都可透过以不同的名称（每个不同的名字借由符号链接或硬链接支持）调用单一 BusyBox 二进制程序来访问。

    ```
    /bin/busybox ls
    ```

  - 或通过软链接至 Busybox 文件， BusyBox 会侦测其被链接时的名称，并执行对应的指令。

    ```
    /bin/ls -> /bin/busybox
    /binls
    ```

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Busybox 文件系统制作
  - Busybox 也是一套需要从源码编译的软件系统，也需要 make defconfig 以及 make menuconfig ，一些编译参数基本也与编译内核的保持一致
  - 在 menuconfig 中，需要开启几个选项：

```
→ Settings
  --- Support --long-options
    [*] Don't use /usr    #不要生成 usr 目录 (CONFIG_INSTALL_NO_USR)
  --- Build Options
    [*] Build static binary (no shared libs)  #进行静态编译  (CONFIG_STATIC)
  --- Installation Options ("make install" behavior)
    What kind of applet links to install (<choice> [=y])
      (X) as soft-links    (CONFIG_INSTALL_APPLET_SYMLINKS)
  --- Library Tuning
    [*]   Query cursor position from terminal    (CONFIG_FEATURE_EDITING_ASK_TERMINAL)
```

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Busybox 文件系统制作
  - 使用 make 编译 busybox ，立刻报错!

```
...
  LINK    busybox_unstripped
Static linking against glibc, can't use --gc-sections
...
==========
/usr/bin/ld: cannot find -lcrypt: No such file or directory
collect2: error: ld returned 1 exit status
Note: if build needs additional libraries, put them in CONFIG_EXTRA_LDLIBS.
Example: CONFIG_EXTRA_LDLIBS="pthread dl tirpc audit pam"
make: *** [Makefile:718: busybox_unstripped] Error 1
```

  - 但这不是因为我的系统没有这些链接库，据说是因为 glibc 在处理静态链接库时效果不是很好，已经有许多人在近年反馈过编译 busybox 出现该问题

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Busybox 文件系统制作
  - 解决方案是换用 musl 的 C 标准库
    - musl，一种 C 标准函式库，主要使用于以 Linux 内核为主的作业系统上，目标为嵌入式系统与行动装置，采用 MIT 许可证释出．作者为瑞奇·费尔克（Rich Felker）．开发此库的目的是写一份干净、高效、符合标准的 C 标准库．[3]
    - Musl 是从零开始设计的．一是希望让静态链接更高效；二是现有的 C 标准库在一些极端条件下表现很糟糕，竞态条件、资源不足时常常会出问题，而 Musl 试图避免它们来达到较高的实时强健性．[3]Musl 的动态运行时只有一个文件，有稳定的 ABI，因此可以实现无竞态的版本升级．对静态链接的支持也让可移植单文件应用部署成为可能，而且不会使文件体积膨胀很多．
    - Musl 声称与 POSIX 2008 标准和 C11 标准兼容，[4] 还实现了多数广泛使用但非标准的 Linux、BSD 和 glibc 函数．

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- Busybox 文件系统制作
  - 解决方案是换用 musl 的 C 标准库
    - 安装 musl-gcc
    - 使用 make CC=musl-cc 的方式重新编译 busybox！
    - 随后 make CC=musl-cc install 生成 busybox 给出的根目录文件夹，改名为 rootfs
  - 对 rootfs 做一些初始化工作：

```
pushd rootfs
touch init;     cat > init << EOF
#!/bin/busybox sh
mount -t proc none /proc
mount -t sysfs none /sys
exec /sbin/init
EOF
chmod a+x init
find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../rootfs.cpio.gz
popd
```

02 QEMU 与内核编译 – 制作 initrd 根文件系统

- 使用该根目录系统启动 QEMU：

```
rqdmap in ~/KML via v12.2.1-gcc
[ 22:45:21 ] qemu-system-x86_64 -kernel ./bzImage -initrd rootfs.cpio.gz
```

Machine View

```
/ # ls
bin   dev   init  proc  root  sbin  sys
/ # uname -h
uname: unrecognized option: h
BusyBox v1.33.2 (2023-03-06 20:45:51 CST) multi-call binary.

Usage: uname [-amnrspvio]

Print system information

        -a      Print all
        -m      The machine (hardware) type
        -n      Hostname
        -r      Kernel release
        -s      Kernel name (default)
        -p      Processor type
        -v      Kernel version
        -i      The hardware platform
        -o      OS name
/ # uname -v
#4 SMP Mon Mar 6 20:17:26 CST 2023
/ # uname -a
Linux (none) 4.4.12-kml-rt18+ #4 SMP Mon Mar 6 20:17:26 CST 2023 x86_64 GNU/Linu
x
/ #
```

02 QEMU 与内核编译 - 使用 QEMU 调试内核

- 修改 QEMU 启动指令

```
qemu-system-x86_64 -kernel ./bzImage -initrd
rootfs.cpio.gz \
    -append "nokaslr console=ttyS0" -nographic -S -
```

  - -s 等价于 -gdb tcp::1234 表示监听 1234 端口，用于 gdb 连接
  - -S 表示加载后立即暂停，等待调试指令。不设置这个选项内核会直接执行

- 为了方便 GDB 的启动， 在工作目录下创建一个 .gdbinit 文件， 填入：

```
file linux-3.18.140/vmlinux
target remote:1234
```

- 这样， 运行完 QEMU 启动内核，再在工作目录下运行 gdb 即可开始调试！

03 QEMU+KML 启用

- 在上一章节我们介绍了使用 QEMU+Busybox 编译、启动、调试 Linux 内核的一般过程。
- 本章将主要介绍如何真正部署 KML 补丁并验证其已经正常工作的步骤
  - 由于 KML 相关补丁年老失修，是一个比较旧的补丁，且没有开发者进一步跟进与维护，在实际部署中遇到了许多的困难，这些问题与解决方案将一并作为经验进行分享。

03 QEMU+KML 启用 – 如何判断程序运行状态?

- 在开始之前，先考虑一下如何判断程序目前的运行状态?
- 我主要收集到两种办法:
  - 通过寄存器的内容；比如在 x86 架构下通过 cs(代码段寄存器) 访问段选择符的低 2 位， 即可得知请求者的特权级； 或者在 arm 架构下通过 cpsr 寄存器也可以知道其所处的模式． 具体的读取方式用一个 C 的内联汇编进行读取即可。
  - 通过 awk ‘{print 14, 15}’ /proc/<pid>/stat 动态读取程序运行时间，输出的两个数字分别代表用户态和内核态中程序执行的时间． 这个方法其实很不错， 因为通过内联汇编访问寄存器有诸多不便， 特别是 aarch64 好像还把一些寄存器给隐藏起来了不对用户可见... 总之，如何判断 arm 架构下程序是否在内核态还挺费劲， 不如这个方法使用该方法查看

03 QEMU+KML 启用 – 如何使用 KML 技术?

- 如何启用一个 KML 用户程序?
  - /trusted 下的可执行文件默认都将以特权级执行
    - 需要在没有进行 chroot 的情况下，不然可能会造成安全性破坏

03 QEMU+KML 启用 – 版本选择

- 选择什么版本的内核 / 软件版本比较重要， 这里我做了很多次尝试:
  - Github 仓库 KML 补丁， Linux-4.4.12
    - x86_64
    - Aarch64 与 arm
  - 官方版本的 KML 补丁，最新维护到 Linux-4.0.4
    - I386
    - X86_64

03 QEMU+KML 启用 - Github 仓库 KML 补丁 - x86_64

- 最开始尝试的是 Github 上给出的 KML 补丁
  - https://github.com/sonicyang/KML
- 相比于官网， 其最大的特点是支持的内核版本较新， 但也仅支持一个版本：Linux-4.4.12； 而官网的版本则一直维护到 Linux-4.0

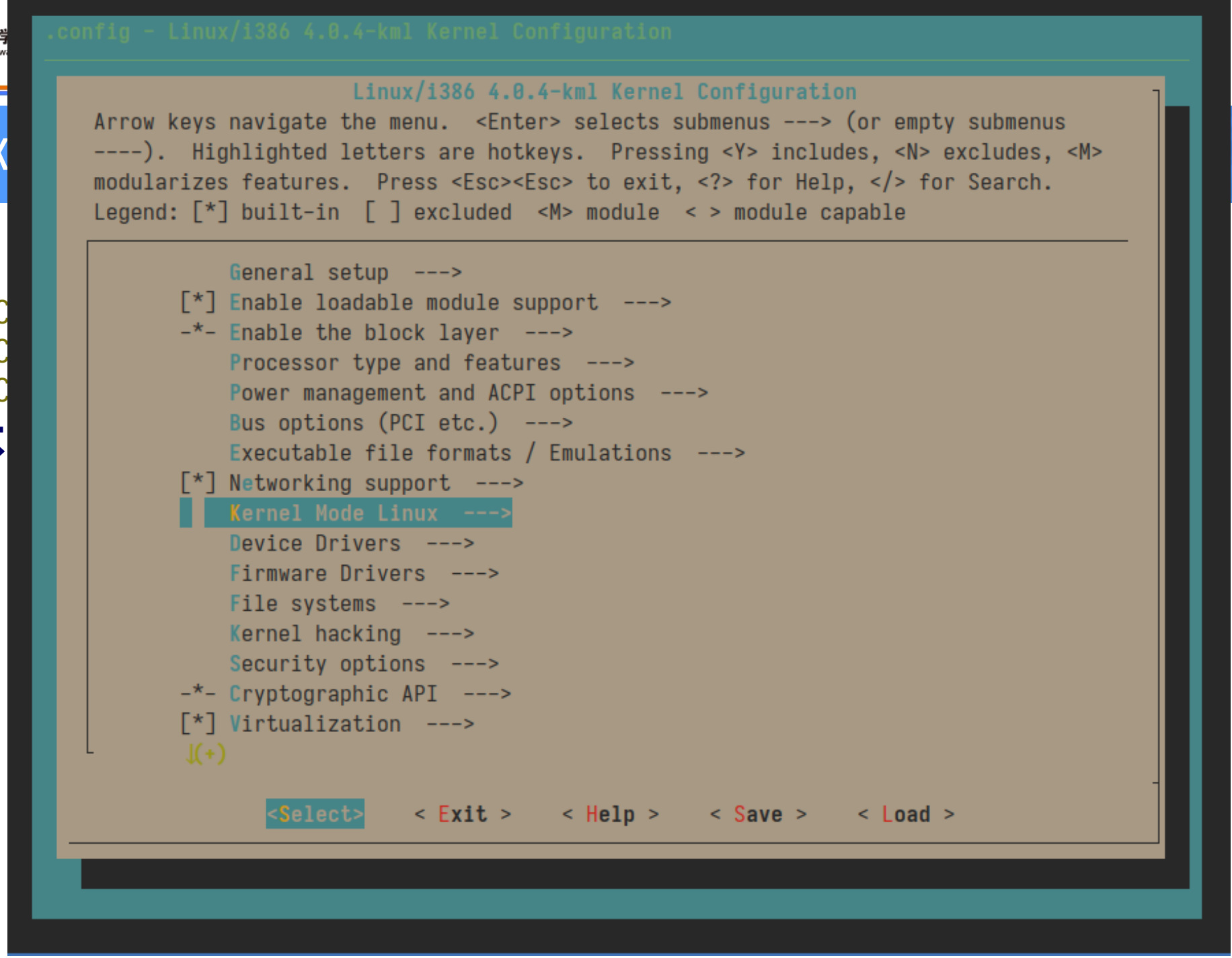
- 第一次上手修改时，没有想太多，直接在编译了本机架构对应的 x86_64 版本，发现 menuconfig 中甚至没有 KML 的任何选项。
- 查看补丁内容后发现该份补丁只对 ARM 架构做了修改!
  - grep

03 QEMU+KML 启用 - Github 仓库 KML 补丁 - ARM

- 下面就开始尝试做交叉编译了，目标是 arm 架构
  - 期间还做了一次 aarch64 的交叉编译， 不过 aarch64 和 arm 是两个架构，aarch64 并不向后兼容 arm； arm64 通常才是 aarch64， 而 arm 好像单纯的只是 arm32
- 为了交叉编译，需要安装对应平台的编译器：
  - Arch Linux - aarch64-linux-gnu-gcc 12.2.0-1 (x86_64)
  - AUR (en) - arm-linux-gnueabihf-gcc

03 QEMU+K

- 以 arm
  - make ARC
  - make ARC
  - make ARC
- 对于不



03 QEMU+KML 启用 - Github 仓库 KML 补丁 - ARM

- **在交叉编译 ARM 时遇到一些问题：**

```
usr/bin/ld: scripts/dtc/dtc-parser.tab.o:(.bss+0x10): multiple definition of
'yylloc'; scripts/dtc/dtc-lexer.lex.o:(.bss+0x0): first defined here

collect2: error: ld returned 1 exit status
```

- ❖ **修改为 extern 避免重复定义：**

```
--- a/scripts/dtc/dtc-lexer.lex.c_shipped
+++ b/scripts/dtc/dtc-lexer.lex.c_shipped
@@ -637,7 +637,7 @@ char *yytext;
 #include "srcpos.h"
 #include "dtc-parser.tab.h"

-YYLTYPE yylloc;
+extern YYLTYPE yylloc;
```

03 QEMU+KML 启用 - Github 仓库 KML 补丁 - ARM

- 在交叉编译 ARM 时遇到一些问题：
  - PIC 相关，在 Makefile 中加入 -fno-PIC 即可：

```
cc1: sorry, unimplemented: code model 'large' with '-fPIC'
make[1]: *** [scripts/Makefile.build:264: crypto/echainiv.o] Error 1
make: *** [Makefile:949: crypto] Error 2
make: *** Waiting for unfinished jobs....
```

03 QEMU+KML 启用 - Github 仓库 KML 补丁 - ARM

- 由于是交叉编译，因而 Busybox 也需要使用交叉编译工具链重新编译！
  - 具体的指令与内核编译类似，通过修改 CROSS_COMPILE= 以及 ARCH= 两个参数指定编译工具和目标架构。
- 最终使用 QEMU 进行启动：

```
qemu-system-arm  -M  virt -cpu cortex-a53 -smp 2 -m 4096M \
    -kernel ./Image.gz   -nographic \
    -append "console=ttyAMA0 init=/linuxrc ignore_loglevel" \
    -initrd ./rootfs.cpio.gz
```

  - 与 x86_64 不同， qemu-arm 必须要指定 -M(machine)，-cpu 等参数才可成功运行。
- 坏消息是， ARM 架构的内核不知为何无法启用 KML 机制，特殊目录下的程序特权级仍为 3；由于对 ARM 架构完全不算熟悉，因而放弃调试。

03 QEMU+KML 启用 - 官方版本 KML - i386

- Github 仓库的版本支持过于单一，因而打算切换至 KML 作者官方给出的补丁进行尝试。在选择版本时，需要同时考虑到 KML+RT 补丁的支持情况，同时尽可能选择较新的内核版本。
  - 满足上述条件的最新版本是 Linux-4.0.4(这也是 8 年前了 !)， 但更老的我也做了尝试， 主要有几个坑点：
    - 一些版本 (主要是老版本才有的) 的 include/linux/compiler-gcc.h 文件会检查 gcc 版本， 以 4.0.4 为例，其最高只支持到 gcc-5 ，需要单独安装旧版 gcc 才可以编译。

```
#define _gcc_header(x) __gcc_header(linux/compiler-gcc##x.h)
#define gcc_header(x) _gcc_header(x)
#include gcc_header(__GNUC__)

ll include/linux/ | grep gcc
-rw-r--r--  1 rqdmap rqdmap    635 May 18  2015 compiler-gcc3.h
-rw-r--r--  1 rqdmap rqdmap   3089 May 18  2015 compiler-gcc4.h
-rw-r--r--  1 rqdmap rqdmap   2484 May 18  2015 compiler-gcc5.h
-rw-r--r--  1 rqdmap rqdmap   4280 May 18  2015 compiler-gcc.h
```

03 QEMU+KML 启用 - 官方版本 KML - i386

- Github 仓库的版本支持过于单一，因而打算切换至 KML 作者官方给出的补丁进行尝试。在选择版本时，需要同时考虑到 KML+RT 补丁的支持情况，同时尽可能选择较新的内核版本。
  - 满足上述条件的最新版本是 Linux-4.0.4(这也是 8 年前了 !)， 但更老的我也做了尝试， 主要有几个坑点：
    - 一些版本 (主要是老版本才有的) 的 include/linux/compiler-gcc.h 文件会检查 gcc 版本， 以 4.0.4 为例，其最高只支持到 gcc-5 ，需要单独安装旧版 gcc 才可以编译。
    - 在架构选择上首先使用 i386 做了尝试，认为其适用性可能更广：

```
make ARCH=i386 CC=gcc-4.4 defconfig
make ARCH=i386 CC=gcc-4.4 menuconfig
make ARCH=i386 CC=gcc-4.4 -j 20
```

03 QEMU+KML 启用 - 官方版本 KML - i386

- Github 仓库的版本支持过于单一，因而打算切换至 KML 作者官方给出的补丁进行尝试。在选择版本时，需要同时考虑到 KML+RT 补丁的支持情况，同时尽可能选择较新的内核版本。
  - i386 架构也需要配备 i386 的 rootfs， 因而需要为 usybox 编译一下 i386 的版本
    - 此前静态链接编译 busybox 时 glibc 处理不力，这里也是一样， 需要使用 i386 的 musl 进行编译！ 为此安装 kernel-headers-musl-i386 以及 i386-musl 包， 即可获得 i386-musl- 工具链
    - busybox 的 menuconfig 中需要指定一些参数， 使得编译出来的是 32 位的程序：

```
Busybox Settings  --->
    Build Options  --->
        [*] Build BusyBox as a static binary (no
shared libs)
        (-m32 -march=i386 -mtune=i386) Additional
CFLAGS
        (-m32) Additional LDFLAGS
```

03 QEMU+KML 启用 - 官方版本 KML - i386

- 最终为 i386 架构编译完成， 但无法引导进去系统 .
- 其问题是内核加载的过程中不知道哪里出错了， 结果导致无法进入 .
- 下面截图做了个对比:
  - 其中左边是将 busybox 本体文件放在了 rootfs 的 /trusted 下， /bin 下的是指向该位置的软链接；
  - 而右边的是 busybox 本体就在 /bin 下， 没有放任何东西在 /trusted.
  - 好消息？ 左右两边的内核是一样的， 区别只是有没有将所有的应用（因为所有的应用其实都是链接到 busybox 的）， 明显可以看到有没有 trusted 对与内核日志的输出有很大的变化 . 这可能说明在这里 trusted 确实起到了作用， 只要修一下为什么 i386 架构的内核无法启动可能即可开箱即用 kml 机制 .

03 QEMU+KML 启用 - 官方版本 KML - i386


```
[    0.854376]   Magic number: 15:302:788
[    0.854687] console [netcon0] enabled
[    0.854735] netconsole: network logging started
[    0.855789] ALSA device list:
[    0.855828]   No soundcards found.
[    0.935101] Freeing unused kernel memory: 688K (c1ab1000 - c1b5d000)
[    0.938600] modprobe (870) used greatest stack depth: 6572 bytes left
[    0.939554] Write protecting the kernel text: 7912k
[    0.939707] Write protecting the kernel read-only data: 2540k
[    0.947893] BUG: unable to handle kernel paging request at cf18fff0
[    0.948043] IP: [<c17b80cb>] page_fault+0x1b/0x38
[    0.948043] *pde = 00000000
[    0.948043] Oops: 0002 [#1] SMP
[    0.948043] Modules linked in:
[    0.948043] CPU: 0 PID: 1 Comm: init Not tainted 4.0.4-kml-rt1+ #1
[    0.948043] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.1-1-1 04/01/2014
[    0.948043] task: c78a4000 ti: c789e000 task.ti: c789e000
[    0.948043] EIP: 0060:[<c17b80cb>] EFLAGS: 00040493 CPU: 0
[    0.948043] EIP is at page_fault+0x1b/0x38
[    0.948043] EAX: b77fb800 EBX: b77fb8ac ECX: b77fb800 EDX: 08166380
[    0.948043] ESI: 08164ff4 EDI: 00000000 EBP: bffabcac ESP: c789fff4
[    0.948043]  DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
[    0.948043] CR0: 8005003b CR2: cf18fff0 CR3: 07253000 CR4: 00000690
[    0.948043] Stack:
[    0.948043]  000003ff 00000000 00000000
[    0.948043] Call Trace:
[    0.948043] Code: eb 46 66 90 8d 76 00 68 40 48 04 c1 eb 3a 66 90 81 fc 00 00 00 c0 77 25 89 6c 24 08 89 e5 64 8b 0
[    0.948043] EIP: [<c17b80cb>] page_fault+0x1b/0x38 SS:ESP 0068:c789fff4
[    0.948043] CR2: 00000000cf18fff0
[    0.948043] ---[ end trace 5b2999e00c2fd02c ]---
[    0.965409] Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000009
[    0.965409]
[    0.965721] Kernel Offset: 0x0 from 0xc1000000 (relocation range: 0xc0000000-0xc87dffff)
[    0.965807] ---[ end Kernel panic - not syncing: Attempted to kill init! exitcode=0x00000009
[    0.965807]
QEMU: Terminated
rqdmap in ~/KML via v12.2.1-gcc took 3s
[ 22:46:01 ]
rqdmap in ~/KML via v12.2.1-gcc
[ 22:46:01 ]
rqdmap in ~/KML via v12.2.1-gcc
[ 22:47:34 ] trusted
```

```
[    0.851540]   Magic number: 15:302:788
[    0.851855] console [netcon0] enabled
[    0.851905] netconsole: network logging started
[    0.852988] ALSA device list:
[    0.853061]   No soundcards found.
[    0.931543] Freeing unused kernel memory: 688K (c1ab1000 - c1b5d000)
[    0.939367] modprobe (870) used greatest stack depth: 6608 bytes left
[    0.940362] Write protecting the kernel text: 7912k
[    0.940518] Write protecting the kernel read-only data: 2540k
[    0.965043] traps: init[889] general protection ip:8131a5a sp:bfce1c24 error:0 in busybox[8049000+ea000]
[    0.967100] Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b
[    0.967100]
[    0.967318] CPU: 0 PID: 1 Comm: init Not tainted 4.0.4-kml-rt1+ #1
[    0.967373] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.1-1-1 04/01/2014
[    0.967520]  c78a4000 c789fdf4 c17b0fa4 c789fe4c c789fe14 c17b0cc1 c1958c74 c1b74980
[    0.967626]  00000000 c789fe4c c78a4000 c78ac00c c789fe64 c10515c0 c1958f04 0000000b
[    0.967712]  c1075fe6 395862d7 c78a450e 00000001 c789ff10 c789fe44 c78ac00c c71ad73c
[    0.967820] Call Trace:
[    0.968045]  [<c17b0fa4>] dump_stack+0x41/0x55
[    0.968045]  [<c17b0cc1>] panic+0x7d/0x1a1
[    0.968045]  [<c10515c0>] do_exit+0xaf0/0xb10
[    0.968045]  [<c1075fe6>] ? set_next_entity+0xb6/0xd0
[    0.968045]  [<c1051619>] do_group_exit+0x39/0xa0
[    0.968045]  [<c105bb6b>] get_signal+0x1eb/0x570
[    0.968045]  [<c1001e7b>] do_signal+0x1b/0xb40
[    0.968045]  [<c17b197c>] ? schedule+0x3c/0xa0
[    0.968045]  [<c17b3b1d>] ? schedule_timeout+0x10d/0x180
[    0.968045]  [<c107685f>] ? check_preempt_wakeup+0xff/0x230
[    0.968045]  [<c17b23f5>] ? wait_for_completion_killable+0xc5/0x100
[    0.968045]  [<c104e2b0>] ? do_fork+0xd0/0x2b0
[    0.968045]  [<c10029d8>] do_notify_resume+0x38/0x44
[    0.968045]  [<c17b4ab1>] work_notifysig+0x24/0x2b
[    0.968045] Kernel Offset: 0x0 from 0xc1000000 (relocation range: 0xc0000000-0xc87dffff)
[    0.968045] ---[ end Kernel panic - not syncing: Attempted to kill init! exitcode=0x0000000b
[    0.968045]
QEMU: Terminated
rqdmap in ~/KML via v12.2.1-gcc took 2s
[ 22:46:13 ]
rqdmap in ~/KML via v12.2.1-gcc
[ 22:46:21 ]
rqdmap in ~/KML via v12.2.1-gcc
[ 22:47:34 ] no-trusted
```

03 QEMU+KML 启用 - 官方版本 KML - i386

- 但这里可以确定的是， 不是 KML 模块的问题， 也好像不是 rootfs 的问题
  - 我为了控制变量， 专门编译了一个完全没有 RT 和 KML 补丁的内核， 同样也无法启动进入内核 .
  - 如果没有文件系统的话内核正常也无法启动， 我为其分配了一个 hello kernel 的 initrd 后也无法启动， 因而大概可以排除是 busybox 的锅， 也就不是文件系统的锅 .
- 总体感觉还是内核过老， 与现在新版本的某些东西起了冲突...
- Linux Kernel... （不确定是否是 QEMU 模拟的问题）
  - On 11 December 2012, Torvalds decided to reduce kernel complexity by removing support for i386 processors, making the 3.7 kernel series the last one still supporting the original processor.[60] [61] The same series unified support for the ARM processor.[62]

03 QEMU+KML 启用 - 官方版本 KML - x86_64

- 最终还是决定调试 x86_64 架构，并且选用较新的 linux-3.18.140
  - 关于 Linux 版本号， linux-x.y.z 中 x 和 y 是大小版本号， z 是修订号；linux-3.18.1 首发于 14 年 12 月， 但是最新的 linux-3.18.140 一直到 19 年 5 月，这也导致了该版本的内核较新，可以引导启动
  - 另一个有关的地方是 patch． 直接打 kml 的 3.18 的补丁并不能全部匹配上， 经过测试发现主要有 4 个 rej 文件， 这里需要逐行去对照修改！ 其中代码的顺序或位置可能发生了较大的变化， 但是经过测试， 只要将这些语句照着 kml 的补丁改过来即可， 不需要考虑位置的问题．

```
10848  2023-03-09 19:41  vim arch/x86/kernel/cpu/common.c.rej
10852  2023-03-09 19:44  vim arch/x86/syscalls/syscall_32.tbl.rej
10888  2023-03-09 20:06  vim arch/x86/kernel/head64.c.rej
10889  2023-03-09 20:06  vim fs/binfmt_elf.c.rej
```

03 QEMU+KML 启用 - 官方版本 KML - x86_64

- 关于补丁：
  - 在上面提及的 4 个存在冲突的补丁中， syscall_32.tbl.rej 记录了系统调用表，其具有上百条记录。一旦出现了冲突， .rej 文件会将其全部打印出来，根本无法定位到具体冲突的行号
  - 我对补丁不够熟悉，目前使用了这样一种方法来找到冲突位置：
    - :%g/^+//　用 Vim 全局命令即可快速删除以 + 开头的行
    - 再用多列选择把其余行开头的 - 号删掉
    - 最后使用 diff 比较其与目前的文件的区别即可 .

```
# diff syscall_32.tbl syscall_32.tbl.rej
297c297
< 288    i386    keyctl          sys_keyctl
 compat_sys_keyctl
---
> 288    i386    keyctl          sys_keyctl
```

03 QEMU+KML 启用 - 官方版本 KML - x86_64

- 修改完所有的补丁，即可在 QEMU 中体验到 KML 技术。

03 QEMU+KML 启用 - 官方版本 KML - x86_64

```
[    0.859583] Write protecting the kernel read-only data: 14336k

[    0.863326] Freeing unused kernel memory: 1820K
[    0.869909] Freeing unused kernel memory: 1380K
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console. / #
/ # [    1.337459] tsc: Refined TSC clocksource calibration: 2688.098 MHz
[    1.393017] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042/serio1/input/input3
[    2.339543] Switched to clocksource tsc

/ #
/ # cat main.c
#include <stdio.h>
#include <stdint.h>

int main() {
        uint32_t cs;
        asm volatile("mov %%cs, %0" : "=r" (cs));
        printf("Privilege level: %x\n", cs & 0x3);
        fflush(stdout);
        return 0;
}
/ # ./main
Privilege level: 3
/ # /trusted/main
Privilege level: 0
/ # QEMU: Terminated
rqdmap in ~/KML via v12.2.1-gcc took 15s
```

读取cs寄存器低2位，获取当前运行状态

一般的程序只能运行在用户态(3)

/trusted目录下的程序默认运行在内核态(0)

04 KML 技术内幕

- 本章将参考作者给出的介绍、 Linux 内核源码以及 KML 补丁代码对 KML 技术的实现内幕尝试进行分析，并主要介绍：
  - 如何为用户程序提权?
  - 如何处理 Doublefault 异常

04 KML 技术内幕 - 程序提权

- KML 最显著的特点就在于其可以在内核模式下执行用户程序，那么对其原理的分析也应该从这里开始。
- 据官方 Guide 所说， KML 机制用一个特殊的 start_kernel_thread 例程，其会被 execve 系统调用调用，并设置一系列特殊的寄存器。
  - 内核原生的例程是 start_thread， 其将 CS 段寄存器的内容设置为 __USER_CS;
  - 而 start_kernel_thread 设置为 __KERNEL_CS， 因此 KML 程序将在内核态下运行

04 KML 技术内幕 - 程序提权

- 为了较为清晰地看到程序执行所在的位置，我们通过 gdb 截取调用 start_kernel_thread 时的调用栈

```
(gdb) bt
#0  start_kernel_thread (regs=0xffff88000720ff58, new_ip=4199856, new_sp=140731794638672)
at arch/x86/kernel/process_64.c:271
#1  0xffffffff811b5f30 in load_elf_binary (bprm=0xffff8800065a4500) at
fs/binfmt_elf.c:1039
#2  0xffffffff8116b76a in search_binary_handler (bprm=0xffff8800065a4500) at
fs/exec.c:1425
#3  0xffffffff8116d21f in exec_binprm (bprm=<optimized out>) at fs/exec.c:1467
#4  do_execve_common (filename=0xffff880000014000, argv=..., envp=...) at fs/exec.c:1564
#5  0xffffffff8116d482 in do_execve (filename=0xffff88000720ff58,
__argv=0xffff880006af9200, __envp=0x25b4354a) at fs/exec.c:1606
#6  0xffffffff8116d4ba in SYSC_execve (envp=<optimized out>, argv=<optimized out>,
filename=<optimized out>) at fs/exec.c:1660
#7  SyS_execve (filename=<optimized out>, argv=140096697936064, envp=140096697936080) at
fs/exec.c:1658
#8  0xffffffff818311eb in stub_execve () at arch/x86/kernel/entry_64.S:787
```

04 KML 技术内幕 - 程序提权

- 当执行一个可执行文件时， 除了 execve() 以外的 exec 类函数都是 C 库的封装例程， 其本质都调用了 execve() 系统调用 .
- 而该系统调用 sys_execve 会实质调用 do_execve 函数， 其将在执行文件前做一系列准备 (比如把文件路径名、命令行参数、环境串等拷贝到一个或多个新分配的页框中， 填充 linux_binprm 结构等)
- 随后检查 formats 链表 (这个链表是一个存放了所有 linux_binfmt 对象的单向链表)， 尽力将 linux_binprm 传递给链表元素的 load_binary 方法； 一旦成功应答， 则对 formats 扫描结束 .

04 KML 技术内幕 - 程序提权

- Linux 下的可执行文件格式通常是 ELF ，该格式本质上提供了 3 种方法：load_binary，load_shlib，core_dump.
- 我们这里主要关注 load_binary 方法， 其通过读取存放在可执行文件中的信息为当前进程建立一个新的执行环境 .

```
static struct linux_binfmt elf_format = {
        .module         = THIS_MODULE,
        .load_binary    = load_elf_binary,
        .load_shlib     = load_elf_library,
        .core_dump      = elf_core_dump,
        .min_coredump   = ELF_EXEC_PAGESIZE,
};
```

04 KML 技术内幕 - 程序提权

- 在 load_binary 中存储了 load_elf_binary 这个函数，该函数有点长，大概有 400 多行… 所以我们不去全部关注， 只看一下 KML 修改的部分：
  - 该函数会首先检查目前是否处于安全的 KML 状态，通过调用 is_safe 来进行判断
  - 随后根据是否启用了 KML mode 来决定使用哪个内核例程对该程序将要使用的寄存器进行修改

```
static int load_elf_binary(struct linux_binprm *b
{
#ifdef CONFIG_KERNEL_MODE_LINUX
        kernel_mode = is_safe(bprm->file);
#endif
        ...
#ifndef CONFIG_KERNEL_MODE_LINUX
        start_thread(regs, elf_entry, bprm->p);
#else
        if (kernel_mode) {
                start_kernel_thread(regs, elf_ent
        } else {
                start_thread(regs, elf_entry, bpr
        }
#endif
```

- 下面将看一下这两个相关函数

04 KML 技术内幕 - 程序提权 - fs/binfmt_elf.c:is_safe

```
#include <linux/fs_struct.h>
#define TRUSTED_DIR_STR         "/trusted/"
#define TRUSTED_DIR_STR_LEN     9

static inline int is_safe(struct file* file)
{
        int ret;
        char* path;
        char* tmp;

#ifdef CONFIG_KML_CHECK_CHROOT
        if (current_chrooted()) {
                return 0;
        }
#endif
        tmp = (char*)__get_free_page(GFP_KERNEL);

        if (!tmp) {
                return 0;
        }

        path = d_path(&file->f_path, tmp,
PAGE_SIZE);
        ret = (0 == strncmp(TRUSTED_DIR_STR, path,
TRUSTED_DIR_STR_LEN));

        free_page((unsigned long)tmp);
        return ret;
}
```

- current_chrooted 是内核原生函数，用于检查是否在 chroot 环境中
- d_path 用于将一个目录入口转换为 ASCII 的路径名，并存储在缓冲区 tmp 中

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

```
void
start_thread(struct pt_regs *regs, unsigned long
new_ip,
        unsigned long new_sp)
{
        set_user_gs(regs, 0);
        regs->fs                = 0;
        regs->ds                = __USER_DS;
        regs->es                = __USER_DS;
        regs->ss                = __USER_DS;
        regs->cs                = __USER_CS;
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->flags             = X86_EFLAGS_IF;
```

```
#ifdef CONFIG_KERNEL_MODE_LINUX
void
start_kernel_thread(struct pt_regs *regs, unsigned
long new_ip,
        unsigned long new_sp)
{
        set_user_gs(regs, 0);
        regs->fs                = __KERNEL_PERCPU;
        set_fs(KERNEL_DS);
        regs->ds                = __USER_DS;
        regs->es                = __USER_DS;
        regs->ss                = __KERNEL_DS;
        regs->cs                = __KU_CS_EXCEPTION;
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->flags             = X86_EFLAGS_IF;
...
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

- KML 重写了一份 start_kernel_thread 函数； 值得注意，start_kernel_thread 以及 start_thread 函数其实位于 load_elf_binary 末尾位置， 这意味着其之前所做的所有其余工作与执行一个一般的可执行文件完全相同．
- 这使得我们对这段代码的分析可以不用涉及太多加载可执行文件的其余的过程。

```
#ifdef CONFIG_KERNEL_MODE_LINUX
void
start_kernel_thread(struct pt_regs *regs, unsigned
long new_ip,
        unsigned long new_sp)
{
        set_user_gs(regs, 0);
        regs->fs                = __KERNEL_PERCPU;
        set_fs(KERNEL_DS);
        regs->ds                = __USER_DS;
        regs->es                = __USER_DS;
        regs->ss                = __KERNEL_DS;
        regs->cs                = __KU_CS_EXCEPTION;
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->flags             = X86_EFLAGS_IF;
...
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

- 这段代码首先也是清理了 gs 寄存器，清理当前线程可能的 TLS(Thread-Local Storage)
- 随后出现了第一个区别，其利用了 fs 寄存器；在 x86 中 fs 寄存器作为辅助段寄存器使用。
  - 可以看到，其不仅直接修改了当前寄存器结构中的 fs 内容，还调用了 set_fs 这样一个奇怪的函数 / 宏

```
#ifdef CONFIG_KERNEL_MODE_LINUX
void
start_kernel_thread(struct pt_regs *regs, unsigned
long new_ip,
        unsigned long new_sp)
{
        set_user_gs(regs, 0);
        regs->fs                = __KERNEL_PERCPU;
        set_fs(KERNEL_DS);
        regs->ds                = __USER_DS;
        regs->es                = __USER_DS;
        regs->ss                = __KERNEL_DS;
        regs->cs                = __KU_CS_EXCEPTION;
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->flags             = X86_EFLAGS_IF;
...
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

- 追踪到 set_fs 宏中看一下，发现其实际上设置的是当前进程的 thread_info 字段中的 addr_limit 内容，该字段限制了当前的进程地址空间。
- 根据注释所说，当 get_fs() 为 KERNEL_DS 时则不进行参数检查，原来 KML 设置该字段是为了跳过系统调用传参时的参数检查，就这个函数名而言这确实是 grossly misnamed!

```
/*
 * The fs value determines whether argument validity checking
should be
 * performed or not.  If get_fs() == USER_DS, checking is
performed, with
 * get_fs() == KERNEL_DS, checking is bypassed.
 *
 * For historical reasons, these macros are grossly misnamed.
 */

#define MAKE_MM_SEG(s)  ((mm_segment_t) { (s) })

#define KERNEL_DS       MAKE_MM_SEG(-1UL)
#define USER_DS         MAKE_MM_SEG(TASK_SIZE_MAX)

#define get_ds()        (KERNEL_DS)
#define get_fs()        (current_thread_info()->addr_limit)
#define set_fs(x)       (current_thread_info()->addr_limit =
(x))
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

- 关于参数检查补充说明一些．比如异常处理程序在执行时就会检查异常发生在用户态还是内核态， 如果处于内核态还会检查是否由于系统调用的无效参数导致， 内核拥有一些手段防御受到无效的系统调用参数的攻击． 出现在内核态的其余异常均是由于内核 Bug 导致， 为了避免硬盘上的数据崩溃处理程序将调用 die() 在控制台打印出所有的 CPU 寄存器内容， 并调用 do_exit 终止当前进程．

```
/*
 * The fs value determines whether argument validity checking
should be
 * performed or not.  If get_fs() == USER_DS, checking is
performed, with
 * get_fs() == KERNEL_DS, checking is bypassed.
 *
 * For historical reasons, these macros are grossly misnamed.
 */

#define MAKE_MM_SEG(s)  ((mm_segment_t) { (s) })

#define KERNEL_DS       MAKE_MM_SEG(-1UL)
#define USER_DS         MAKE_MM_SEG(TASK_SIZE_MAX)

#define get_ds()        (KERNEL_DS)
#define get_fs()        (current_thread_info()->addr_limit)
#define set_fs(x)       (current_thread_info()->addr_limit =
(x))
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/process_32.c:process_32.c

- 回到 start_kernel_thread 代码中，其接着正常的设置了 ds 和 es 字段，再把 ss 设置为 __KERNEL_DS 以启用内核栈
- 随后其为 cs 字段设置了一个神奇的 __KU_CS_EXCEPTION 值， 这并不是原来所想的 KML 机制仅仅只是简单地把该字段设置为 __KERNEL_CS 即可 .
- 而之后的操作与内核原生函数类似，就不多分析了 . 因而下面我们将就 __KU_CS_EXCEPTION 字段进行研究，分析其究竟如何发挥作用， cs 字段如何就被偷偷替换成了 __KERNEL_CS

```
#ifdef CONFIG_KERNEL_MODE_LINUX
void
start_kernel_thread(struct pt_regs *regs, unsigned
long new_ip,
        unsigned long new_sp)
{
        set_user_gs(regs, 0);
        regs->fs                = __KERNEL_PERCPU;
        set_fs(KERNEL_DS);
        regs->ds                = __USER_DS;
        regs->es                = __USER_DS;
        regs->ss                = __KERNEL_DS;
        regs->cs                = __KU_CS_EXCEPTION;
        regs->ip                = new_ip;
        regs->sp                = new_sp;
        regs->flags             = X86_EFLAGS_IF;
...
```

04 KML 技术内幕 - 程序提权 - arch/x86/include/asm/segment.h

- 众所周知，cs 寄存器存放的是 16 位的段选择符，低 2 位确定 CPL(Current Privilege Level)，3-15 位为索引号，用于将程序的逻辑地址通过分段单元变换到线性地址
- KML 利用了一个 u32 的高 16 位，自定义了两个标志位，分别在第 16 和 17 位，用于在随后的内核处理中标记这是一个 KML 用户段。
- 我们关注的 __KU_CS_EXCEPTION 正是置位了第 17 个 bit 位并或上了 __USER_CS 的段描述符

```
#define __USER_DS
(GDT_ENTRY_DEFAULT_USER_DS*8+3)
#define __USER_CS
(GDT_ENTRY_DEFAULT_USER_CS*8+3)

#ifdef CONFIG_KERNEL_MODE_LINUX
#define __KU_CS_INTERRUPT       ((1 << 16) |
__USER_CS)
#define __KU_CS_EXCEPTION       ((1 << 17) |
__USER_CS)

#define kernel_mode_user_process(cs) ((cs) &
0xffff0000)
#define need_error_code_fixup_page_fault(cs)
== __KU_CS_EXCEPTION)
#define __KU_CS         (0x7fff0003 | __KERNEL_CS)
#endif
 regs->cs        = __KU_CS;
```

- 64 位处理器的 start_thread_kernel 中则直接通过设置了高 16 位作为特殊标志？

04 KML 技术内幕 - 程序提权 - arch/x86/include/asm/segment.h

- 那么内核如何利用 __KU_CS_EXCEPTION 标记呢？
- 最显然易见的就是紧随其后的两个宏：
  - kernel_mode_user_process: 仅在 arch/x86/kernel/signal.c 被使用，用于处理 KML 程序正确接受到信号量
  - need_error_code_fix_on_page_fault: 仅在 arch/x86/mm/fault.c 被使用，用于处理缺页中断和页表相关事项
  - 由于这些部分较为庞杂，不予深入研究了

```
#define __USER_DS
(GDT_ENTRY_DEFAULT_USER_DS*8+3)
#define __USER_CS
(GDT_ENTRY_DEFAULT_USER_CS*8+3)

#ifdef CONFIG_KERNEL_MODE_LINUX
#define __KU_CS_INTERRUPT       ((1 << 16) |
__USER_CS)
#define __KU_CS_EXCEPTION       ((1 << 17) |
__USER_CS)

#define kernel_mode_user_process(cs) ((cs) &
0xffff0000)
#define need_error_code_fix_on_page_fault(cs) ((cs)
== __KU_CS_EXCEPTION)
#endif
```

04 KML 技术内幕 - 程序提权 - arch/x86/include/asm/segment.h

- 那么内核如何利用 __KU_CS_EXCEPTION 标记呢？
- 其最终会在系统调用中发挥作用，整条调用链非常的长，且大部分均为汇编指令，因而下面只截取部分进行说明：
  - 汇编宏：MAKE_DIRECTCALL_SPECIAL
  - 汇编入口：system_call

```
#define __USER_DS
(GDT_ENTRY_DEFAULT_USER_DS*8+3)
#define __USER_CS
(GDT_ENTRY_DEFAULT_USER_CS*8+3)

#ifdef CONFIG_KERNEL_MODE_LINUX
#define __KU_CS_INTERRUPT       ((1 << 16) |
__USER_CS)
#define __KU_CS_EXCEPTION       ((1 << 17) |
__USER_CS)

#define kernel_mode_user_process(cs) ((cs) &
0xffff0000)
#define need_error_code_fix_on_page_fault(cs) ((cs)
== __KU_CS_EXCEPTION)
#endif
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/direct_call_32.h

- **汇编宏：MAKE_DIRECTCALL_SPECIAL**
  - 该汇编宏的作用是辅助生成系统调用表
  - 可以看到其最后压入了 2 个 32 位的内容进栈：%cs 以及一段代码地址，随后便无条件跳转进入 system_call 汇编代码段
  - 在 system_call 中将利用刚刚压入栈中的 cs 寄存器中的特殊字段 __KU_CS_EXCEPTION

```
#define MAKE_DIRECTCALL_SPECIAL(entry, argnum,
syscall_num) \
.text; \
ENTRY(direct_ ## entry); \
        pushl %ebx; \
        pushl %edi; \
        pushl %esi; \
        pushl %ebp; \
        add $-4, %esp; \
\
        movl $(syscall_num), %eax; \
\
        call direct_special_work_ ## argnum; \
\
        pushfl; \
        pushl %cs; \
        pushl $direct_wrapper_int_post; \
        jmp system_call;
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/entry_32.S:system_call

- 汇编代码段：system_call
  - 在这段原生代码接近结束的时候，KML 做了修饰．通过比较 6(%esp) 是否为 0 决定是否要跳转到 ret_to_ku 这个 KML 实现的特殊的汇编代码语句中．
  - 这是由于考虑到 MAKE_DIRECTCALL_SPECIAL 的压栈顺序，栈中 cs 寄存器的内容实际占据第 5-8 个字节，访问 6(%esp) 其实就是取了 cs 寄存器的高 16 位．如果其不为 0，说明这是一个 KML 标志过的 cs 段，因为也没有其他内核自身的代码会修改这高 16 位．
- 下面看看 ret_to_ku 又做了什么

```
# system call handler stub
ENTRY(system_call)
    RING0_INT_FRAME
    SWITCH_STACK_TO_KK_EXCEPTION
    ...
#ifdef CONFIG_KERNEL_MODE_LINUX
restore_all_return:
/* Switch stack KK -> KU. */
    /* check whether if stack switch occured or n
*/
    cmpw $0x0, 6(%esp)
    jne ret_to_ku
#endif
    ...
```

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/entry_32.S:system_call

- **汇编代码段：ret_to_ku**
  - 该段汇编代码通过读取了 4(%esp) 实际读取了产生系统调用的程序的 cs 寄存器内容
  - 通过比较，读取状态寄存器内容，其又细分为 jmp 到两个分支中，异常处理和中断处理．而这部分其实是 KML 花了大力气去修补的，涉及内容较多，因而目前只看一下对于一个系统调用，即软中断，即异常来说，程序会如何进行．

```
ENTRY(ret_to_ku)
    ...
    cmpl $__KU_CS_EXCEPTION, 4(%esp)
    je ret_to_ku_from_exception
    jmp ret_to_ku_from_interrupt
```

- **下面进入到 ret_to_ku_from_exception**

04 KML 技术内幕 - 程序提权 - arch/x86/kernel/entry_32.S:system_call

- **代码段：ret_to_ku_from_exception**
  - 修改 cs 寄存器的内容， 同时做了一些诸如切换栈的工作， 函数返回到 system_call 时则一切已经准备就绪！

```
/*
 * The stack layout for ret_to_ku_from_exception:
 *
 * %esp ---> EIP
 *           __KU_CS_EXCEPTION
 *           EFLAGS
 *           ESP
 *           XXX
 *           ...
 */
```

```
ENTRY(ret_to_ku_from_exception)
    movl $__KERNEL_CS, 4(%esp)  /* XCS = __KERNEL_CS */
    pushl %ebp
/* check whether if we can skip iret or not */
    movl 12(%esp), %ebp     /* load EFLAGS to %ebp */
    testl $~(0x240fd7), %ebp
    movl 16(%esp), %ebp     /* load ESP to %ebp */
    jz skip_iret

    addl $-16, %ebp
ret_to_ku_mov_ebp:  popl (%ebp)     /* old EBP */
ret_to_ku_mov_eip:  popl 4(%ebp)        /* EIP */
ret_to_ku_mov_cs:   popl 8(%ebp)        /* XCS */
ret_to_ku_mov_eflags:   popl 12(%ebp)       /* EFLAGS */
            movl %ebp, %esp     /* switch the
```

04 KML 技术内幕 - 程序提权

- 事实上， 在 KML 的 x86 实现中， 主要利用的就是 __KU_CS_EXCEPTION 以及 __KU_CS_INTERRUPT 这两个标志位进行各项处理 .
  - 当带有该特殊标记的 KML 程序运行系统调用时， 内核会动态替换 cs 段的内容为 __KERNEL_CS.
  - 其余诸如信号处理、缺页异常也会利用到该标志位

- KML 对程序提权相关的修改并不继续深入研究了， 由于 KML 对异常处理做的工作也十分的多， 因而下面研究一下 KML 机制下会出现哪些问题以及其是如何应对的 .

04 KML 技术内幕 - 异常处理

- 在实现 KML 技术的时候， 作者说明了几个遇到的较大的困难：
  - Stack starvation
  - Stack switching
  - Interrupt lost
  - Non-maskable interrupt
- 这些问题其实几乎都与如何处理异常有关． 由于 KML 程序直接在内核态下运行， 因而有时对其进行的中断 / 异常处理会变得比较棘手．
- 在具体看到这几个问题之前， 我们先简单看一下当一个中断 / 异常发生时， CPU 会如何工作．

04 KML 技术内幕 - 异常处理 - 硬件处理

- 假定内核已经初始化， 内核运行在保护模式下． 执行一条指令后，cs 和 eip 寄存器会包含下一条指令的逻辑地址． 在此之前， CPU 将检查前一条指令是否产生了中断 / 异常， 如果发生， 则依次执行：
  - 确定与中断或异常关联的向量 i，0≤i≤255
  - 读 idtr 寄存器指向的 IDT 表中的第 i 项， 假定是一个中断门或陷阱门
  - 读 gdtr 获得 GDT 基地址， 并在其中查找 IDT 表项对应的段描述符．
  - 检查中断是否被授权． 读当前特权级 CPL(cs 寄存器的低 2 位) 和段描述符特权级 DPL， 如果 CPL 低于 DPL， 则产生 13 号异常 General protectoin； 对于软中断， 则要进一步检查门描述符的 DPL 是否小于 CPL， 如果小于也将产生 13 号异常， 这是用来避免用户程序访问特殊的陷阱门或中断门．

04 KML 技术内幕 - 异常处理 - 硬件处理

- 假定内核已经初始化， 内核运行在保护模式下． 执行一条指令后，cs 和 eip 寄存器会包含下一条指令的逻辑地址． 在此之前， CPU 将检查前一条指令是否产生了中断 / 异常， 如果发生， 则依次执行：
  - 检查是否发生特权级的变化． 如 CPL 不同于段选择符的 DPL， 则 CPU 必须开始使用新的特权级相关的栈， 通过下述步骤完成这一点：
    - 读 tr 寄存器， 访问当前运行进程的 TSS 段
    - 将与新特权级相关的栈段和栈指针的值载入 ss 和 esp 寄存器， 其原有的值可在 TSS 中找到
    - 在新的栈中保存 ss 和 esp 的旧值， 这些值定义了与旧特权级相关的栈的逻辑地址．
  - 在新的栈中保存 eflags，cs 和 eip 等内容
  - 将 IDT 第 i 项对应的内容装载到 cs 和 eip 寄存器中， 用于跳转到处理程序的逻辑地址上．

04 KML 技术内幕 - 异常处理 - 硬件处理

- 异常处理程序执行完成后将产生一条 iret 指令，iret 指令本质是将 cs，eip，eflags 装载到对应的寄存器中， 将 CPU 控制权交还给此前被中断的程序 .
  - 其中， 如果处理程序的 CPL 不等于 cs 的低 2 位， 则还要重新从栈中装载旧的 ss 和 esp 寄存器， 回到旧的特权级栈中 .
  - 检查 ds，es，fs，gs 等段寄存器， 如果 DPL < CPL 则清空寄存器， 以防止用户态的程序利用这些寄存器访问内核地址空间 .

04 KML 技术内幕 - 异常处理 - Stack starvation

- KML 遇到的 Stack starvatio 问题就可能发生在这个过程中：
  - 通常来说， 当发生异常处理时，CPU 可能会切换特权级栈来处理异常
  - 但是对于一个 KML 程序来说， 当他在自己的用户栈工作时产生一个中断 / 异常，IA-32 CPU 不会将用户栈切换到内核栈进行异常处理， 而是继续直接使用 KML 程序自己的用户栈 .
    - KML 程序在 ring0 执行， 所以一般的中断处理机制不起作用 .
    - KML 优势在于省去了上下文保存和内核栈切换， 但也因此产生新的问题！
  - 但如果在一些情况下，KML 程序的用户栈所在页框还没及时装载进内存，会发生什么呢？ 其产生一个 Page Fault 异常，CPU 为了处理该异常则企图保存当前的一些寄存器内容到栈中， 而当前的 KML 程序栈又不可访问，CPU 又将产生一个 Doublefault 异常！ 最终 CPU 将始终无法获得一块可用的空间用于异常处理

04 KML 技术内幕 - 异常处理 - Stack starvation

- KML 利用了 TSS 段来处理 Stack starvatio 问题 .
  - x86 体系提供了一个特殊的段结构 TSS 来存放硬件上下文， 尽管 Linux 不使用硬件上下文切换，但是要求它为每个 CPU 创建一个 TSS， 一个目的就是当某个 CPU 从用户态切换到内核态时，它可以从 TSS 中获取内核态堆栈的地址
  - 每当进程切换时，内核都将更新 TSS 中的某些字段；因而 TSS 段只反映 CPU 上当前运行的进程的特权级，而不为每个进程都保留 TSS 段。
- 利用该技术， IA-32 不向栈中压入参数， 而是切换到 TSS 这块特殊的段空间中进行异常处理， 原生 Linux 即使用任务门 8 号异常 doublefault.

```
set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
```

- 不过使用 TSS 处理异常的坏处是开销较大， 因而在 KML 的实现中也只对 doublefault 的异常处理使用了 TSS 段 . 由于 doublefault 并不算

04 KML 技术内幕 - 异常处理 - cpu_init

- 下面将结合代码看看 KML 对 TSS 段使用方式 .
- 对 tss 的设置和初始化在 cpu_inti 函数中， 总体来说：
  - 该函数首先取了 2 个 per_cpu 变量的指针， 这两个变量都是 tss_struct 类型的 .
  - 随后对这两个变量进行初始化
  - 将 tss 插入到 gdt 中相应位置
- 我们将深入考察一下前两个部分涉及的代码片段

```
#ifdef CONFIG_KERNEL_MODE_LINUX
    struct tss_struct* doublefault_tss =
&per_cpu(doublefault_tsses, cpu);
    struct tss_struct* nmi_tss = &per_cpu(nmi_tsses, cpu);
#endif

#ifndef CONFIG_KERNEL_MODE_LINUX
#ifdef CONFIG_DOUBLEFAULT
    /* Set up doublefault TSS pointer in the GDT */
    __set_tss_desc(cpu, GDT_ENTRY_DOUBLEFAULT_TSS,
&doublefault_tss);
#endif
#else
    init_doublefault_tss(cpu);
    init_nmi_tss(cpu);
    __set_tss_desc(cpu, GDT_ENTRY_DOUBLEFAULT_TSS,
doublefault_tss);
    __set_tss_desc(cpu, GDT_ENTRY_NMI_TSS, nmi_tss);
```

04 KML 技术内幕 - 异常处理 - struct tss_struct

- 首先简单看一下 tss_struct:
  - x86_tss 存储了有关硬件的内容
  - io_bitmap 为 IO 端口许可权位图
  - 额外的专属内核栈空间

```
struct tss_struct {
    /*
     * The hardware state:
     */
    struct x86_hw_tss   x86_tss;
    unsigned long       io_bitmap[IO_BITMAP_LONGS + 1];

    /*
     * .. and then another 0x100 bytes for the emergency ker
     */
    unsigned long       stack[64];

#ifdef CONFIG_KERNEL_MODE_LINUX
#define KML_STACK_SIZE (8*16)
    char            kml_stack[KML_STACK_SIZE] __attribute__
(16)));
#endif
} ____cacheline_aligned;
```

04 KML 技术内幕 - 异常处理 - struct x86_hw_tss

```
/* This is the TSS defined by the hardware. */
struct x86_hw_tss {
    unsigned short      back_link, __blh;
    unsigned long       sp0;
    unsigned short      ss0, __ss0h;
    unsigned long       sp1;
    /* ss1 caches MSR_IA32_SYSENTER_CS: */
    unsigned short      ss1, __ss1h;
    unsigned long       sp2;
    unsigned short      ss2, __ss2h;
    unsigned long       __cr3;
    unsigned long       ip;
    unsigned long       flags;
    unsigned long       ax;
    unsigned long       cx;
    unsigned long       dx;
    unsigned long       bx;
    unsigned long       sp;
    unsigned long       bp;
    unsigned long       si;
    unsigned long       di;
    unsigned short      es, __esh;
    unsigned short      cs, __csh;
    unsigned short      ss, __ssh;
    unsigned short      ds, __dsh;
    unsigned short      fs, __fsh;
    unsigned short      gs, __gsh;
    unsigned short      ldt, __ldth;
    unsigned short      trace;
    unsigned short      io_bitmap_base;

} __attribute__((packed));
```

04 KML 技术内幕 - 异常处理 - task.c:INIT_TSS

- 以 doublefault_tsses 每 CPU 变量为例， 看一下其中具体设置的内容：
  - 设置了一些关键寄存器的内容
  - 将 ip 设置为 extern 的函数地址 double_fault_tast， 这是一段汇编代码， 当进入到该任务门执行 doublefault 处理时会调用到该函数 .
- 下面进入汇编中研究一下该代码如何工作

```
extern void double_fault_task(void);

#define INIT_DFT {                          \
    .x86_tss = {                            \
        .ss0        = __KERNEL_DS,          \
        .ldt        = 0,                    \
        .fs     = __KERNEL_PERCPU,          \
        .gs     = 0,                        \
        .io_bitmap_base = INVALID_IO_BITMAP_OFFSET, \
        .ip     = (unsigned long) double_fault_task,    \
        .flags      = X86_EFLAGS_SF | 0x2,      \
        .es     = __USER_DS,                \
        .cs     = __KERNEL_CS,              \
        .ss     = __KERNEL_DS,              \
        .ds     = __USER_DS             \
    }                                   \
}

DEFINE_PER_CPU(struct tss_struct, doublefault_tsses) = INIT
```

04 KML 技术内幕 - 异常处理 - entry_32.S:double_fault_task

- 在进入 double_fault_task 汇编代码前， 栈的结构为：
  - %esp → error_code
    pointer to dft_tss
    pointer to normal_tss
  - 该栈中结构其实就是 KML 自定义的 struct dft_stack_struct 结构

```
struct dft_stack_struct {
    unsigned long error_code;
    struct tss_struct* this_tss;
    struct tss_struct* normal_tss;
};
```

```
ENTRY(double_fault_task)
    movl 4(%esp), %edi      # get current TSS.
/* %edi = current_tss */
    movl 8(%esp), %ebx      # get normal TSS.
/* %ebx = prev_tss */
    # get kernel stack.
    kml_get_kernel_stack %ebx, %esi
    movl %esi, %esp
/* From now on, we can use stack. */
    # recreate stack layout as if normal interrupt
    kml_recreate_kernel_stack_layout %ebx
    call_helper prepare_fault_handler,  \
        $double_fault_fixup, %edi, %ebx
    ret_from_task_without_iret %edi, GDT_ENTRY_TSS
    jmp double_fault_task
```

04 KML 技术内幕 - 异常处理 - entry_32.S:double_fault_task

- 在汇编代码中：
  - 先通过 4(%esp) 和 8(%esp) 取了两个 TSS 段的地址，
  - 随后调用汇编宏 kml_get_kernel_stack 用于获取内核栈， 传入参数为第二个 TSS 段的指针
  - 装载进入 %esp， 切换到内核栈
  - 重建内核栈的结构， 传入参数仍然为第二个 TSS 段的指针
  - 调用辅助汇编函数 call_helper 执行一些初始化工作
  - 返回到 TSS 段中
- 下面将说明介绍其中涉及的汇编宏

```
ENTRY(double_fault_task)
    movl 4(%esp), %edi      # get current TSS.
/* %edi = current_tss */
    movl 8(%esp), %ebx      # get normal TSS.
/* %ebx = prev_tss */
    # get kernel stack.
    kml_get_kernel_stack %ebx, %esi
    movl %esi, %esp
/* From now on, we can use stack. */
    # recreate stack layout as if normal interrupt
    kml_recreate_kernel_stack_layout %ebx
    call_helper prepare_fault_handler,  \
        $double_fault_fixup, %edi, %ebx
    ret_from_task_without_iret %edi, GDT_ENTRY_TSS
    jmp double_fault_task
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_get_kernel_stack

- **kml_get_kernel_stack 宏：**
  - 第一行涉及比较两个内容，一个是 __KERNEL_CS，一个是 TSS_CS(\pre_tss)
  - __KERNEL_CS 如下定义：

```
#define GDT_ENTRY_KERNEL_BASE           (12)
#define GDT_ENTRY_KERNEL_CS
(GDT_ENTRY_KERNEL_BASE+0)
#define __KERNEL_CS (GDT_ENTRY_KERNEL_CS*8)
```

```
.macro kml_get_kernel_stack pre_tss, ret
    cmpw $__KERNEL_CS, TSS_CS(\pre_tss)
    jne 1f
    movl TSS_ESP(\pre_tss), \ret
    # If the previous ESP points to kernel-space,
    # we used the kernel stack.
    cmpl $TASK_SIZE, \ret
    jbe 1f
    cmpl $ia32_sysenter_target, TSS_EIP(\pre_tss)
    jne 2f

    # We used the user stack, so needs to load the
    # kernel stack from ESP0 field of TSS.
1:
    movl PER_CPU_VAR(esp0), \ret
2:
.endm
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_get_kernel_stack

Linux's GDT	Segment Selectors
null	0x0
reserved	
reserved	
reserved	
not used	
not used	
TLS #1	0x33
TLS #2	0x3b
TLS #3	0x43
reserved	
reserved	
reserved	
kernel code	0x60 (__KERNEL_CS)
kernel data	0x68 (__KERNEL_DS)
user code	0x73 (__USER_CS)
user data	0x7b (__USER_DS)

Linux's GDT	Segment Selectors
TSS	0x80
LDT	0x88
PNPBIOS 32-bit code	0x90
PNPBIOS 16-bit code	0x98
PNPBIOS 16-bit data	0xa0
PNPBIOS 16-bit data	0xa8
PNPBIOS 16-bit data	0xb0
APMBIOS 32-bit code	0xb8
APMBIOS 16-bit code	0xc0
APMBIOS data	0xc8
not used	
not used	
not used	
not used	
not used	
double fault TSS	0xf8

```
_stack pre_tss, ret
, TSS_CS(\pre_tss)

_tss), \ret
 ESP points to kernel-space,
nel stack.
 \ret

er_target, TSS_EIP(\pre_tss)


r stack, so needs to load the
om ESP0 field of TSS.

esp0), \ret
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_get_kernel_stack

- **kml_get_kernel_stack 宏：**
  - 第一行涉及比较两个内容，一个是 __KERNEL_CS，一个是 TSS_CS(\pre_tss)
  - 而 TSS_CS 则被定义为 76，结合 pre_tss 的结构可知其访问的正是 cs 寄存器的高 16 位

```
.macro kml_get_kernel_stack pre_tss, ret
    cmpw $__KERNEL_CS, TSS_CS(\pre_tss)
    jne 1f
    movl TSS_ESP(\pre_tss), \ret
    # If the previous ESP points to kernel-space,
    # we used the kernel stack.
    cmpl $TASK_SIZE, \ret
    jbe 1f
    cmpl $ia32_sysenter_target, TSS_EIP(\pre_tss)
    jne 2f

    # We used the user stack, so needs to load the
    # kernel stack from ESP0 field of TSS.
1:
    movl PER_CPU_VAR(esp0), \ret
2:
.endm
```

04 KML 技术内幕 - 异常处理 - struct x86_hw_tss

```
/* This is the TSS defined by the hardware. */
struct x86_hw_tss {
    unsigned short      back_link, __blh;
    unsigned long       sp0;
    unsigned short      ss0, __ss0h;
    unsigned long       sp1;
    /* ss1 caches MSR_IA32_SYSENTER_CS: */
    unsigned short      ss1, __ss1h;
    unsigned long       sp2;
    unsigned short      ss2, __ss2h;
    unsigned long       __cr3;
    unsigned long       ip;
    unsigned long       flags;
    unsigned long       ax;
    unsigned long       cx;
    unsigned long       dx;
    unsigned long       bx;
    unsigned long       sp;
    unsigned long       bp;
    unsigned long       si;
    unsigned long       di;
    unsigned short      es, __esh;
    unsigned short      cs, __csh;
    unsigned short      ss, __ssh;
    unsigned short      ds, __dsh;
    unsigned short      fs, __fsh;
    unsigned short      gs, __gsh;
    unsigned short      ldt, __ldth;
    unsigned short      trace;
    unsigned short      io_bitmap_base;

} __attribute__((packed));
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_get_kernel_stack

- **kml_get_kernel_stack 宏：**
  - 第一行涉及比较两个内容，一个是 __KERNEL_CS，一个是 TSS_CS(\pre_tss)
  - 而 TSS_CS 则被定义为 76，结合 pre_tss 的结构可知其访问的正是 cs 寄存器这 16 位
  - 如果判断不等，则可以直接跳转到 1 号标签对应的代码处，将每 CPU 的 esp0(最高权限栈) 塞入 ret 中结束宏调用
  - 如果判断相等，进一步比较 $TASK_SIZE 与之前 tss→esp 的大小关系，$TASK_SIZE 其实就是内核空间的起始位置
  - 如果此前的 esp 已经指向了内核空间也可

```
.macro kml_get_kernel_stack pre_tss, ret
    cmpw $__KERNEL_CS, TSS_CS(\pre_tss)
    jne 1f
    movl TSS_ESP(\pre_tss), \ret
    # If the previous ESP points to kernel-space,
    # we used the kernel stack.
    cmpl $TASK_SIZE, \ret
    jbe 1f
    cmpl $ia32_sysenter_target, TSS_EIP(\pre_tss)
    jne 2f

    # We used the user stack, so needs to load the
    # kernel stack from ESP0 field of TSS.
1:
    movl PER_CPU_VAR(esp0), \ret
2:
.endm
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_get_kernel_stack

- kml_get_kernel_stack 宏：
  - 如果这些都不满足， 那么说明其正处于 ia32_sysenter_target 指令的第一阶段，pre_tss→esp 指向的是 esp1， 因而需要做一些额外的处理
  - 由于对 sysenter 为代表的系统调用不够熟悉， 因而不深入进去研究了
- 总之， 我们可以认为， 通过该宏，我们返回了一个内核栈的指针存放在 ret 参数中

```
.macro kml_get_kernel_stack pre_tss, ret
    cmpw $__KERNEL_CS, TSS_CS(\pre_tss)
    jne 1f
    movl TSS_ESP(\pre_tss), \ret
    # If the previous ESP points to kernel-space,
    # we used the kernel stack.
    cmpl $TASK_SIZE, \ret
    jbe 1f
    cmpl $ia32_sysenter_target, TSS_EIP(\pre_tss)
    jne 2f

    # We used the user stack, so needs to load the
    # kernel stack from ESP0 field of TSS.
1:
    movl PER_CPU_VAR(esp0), \ret
2:
.endm
```

04 KML 技术内幕 - 异常处理 - entry_32.S:double_fault_task

- 通过 movl %esi，%esp，代码切换了当前栈为内核栈，此后的操作均发生在内核栈上
- 下一行涉及的汇编宏是 kml_recreate_kernel_stack_layout

```
ENTRY(double_fault_task)
    movl 4(%esp), %edi      # get current TSS.
/* %edi = current_tss */
    movl 8(%esp), %ebx      # get normal TSS.
/* %ebx = prev_tss */
    # get kernel stack.
    kml_get_kernel_stack %ebx, %esi
    movl %esi, %esp
/* From now on, we can use stack. */
    # recreate stack layout as if normal interrupt
    kml_recreate_kernel_stack_layout %ebx
    call_helper prepare_fault_handler,  \
        $double_fault_fixup, %edi, %ebx
    ret_from_task_without_iret %edi, GDT_ENTRY_TSS
    jmp double_fault_task
```

04 KML 技术内幕 - 异常处理 - entry_32.S:kml_recreate_kernel_stack_layout

- **kml_recreate_kernel_stack_layout :**
  - **该汇编接受一个参数， 也为之前的 tss 指针**
  - **通过做两次比较， 向当前的栈（内核栈）中压入若干个寄存器的内容**
  - **重建内核栈的结构用于后续使用**

```
.macro kml_recreate_kernel_stack_layout pre_tss
    cmpw $__KERNEL_CS, TSS_CS(\pre_tss)
    jne 1f

    movl TSS_ESP(\pre_tss), %eax
    cmpl $TASK_SIZE, %eax
    ja 2f
1:
    pushl TSS_SS(\pre_tss)
    pushl TSS_ESP(\pre_tss)
2:
    pushl TSS_EFLAGS(\pre_tss)
    pushl TSS_CS(\pre_tss)
    pushl TSS_EIP(\pre_tss)
.endm
```

04 KML 技术内幕 - 异常处理 - entry_32.S:double_fault_task

- 随后进入 call_helper 辅助汇编宏中， 该辅助宏接受多个参数：
  - 第一个为即将 call 的函数指针
  - 随后的三个指针均为函数需要的参数
  - 其还会额外将 esp 也压入栈中作为参数传递给该函数

```
.macro call_helper func target_address cur_tss
pre_tss
    pushl %esp
    pushl \pre_tss
    pushl \cur_tss
    pushl \target_address
    call \func
    addl $16, %esp
.endm
```

```
ENTRY(double_fault_task)
    movl 4(%esp), %edi      # get current TSS.
/* %edi = current_tss */
    movl 8(%esp), %ebx      # get normal TSS.
/* %ebx = prev_tss */
    # get kernel stack.
    kml_get_kernel_stack %ebx, %esi
    movl %esi, %esp
/* From now on, we can use stack. */
    # recreate stack layout as if normal interrupt
    kml_recreate_kernel_stack_layout %ebx
    call_helper prepare_fault_handler,  \
        $double_fault_fixup, %edi, %ebx
    ret_from_task_without_iret %edi, GDT_ENTRY_TSS
    jmp double_fault_task
```

04 KML 技术内幕 - 异常处理 - task.c: prepare_fault_handler

- 通过 call_helper 执行了 C 语言函数 prepare_fault_handler:
  - 该函数拥有 asmlinage 标记， 这指示编译器该函数的参数全部从栈中取得， 这与之前 call_helper 压栈的操作是彼此对应的
  - 该函数首先将清零 tss 描述符的第 41 个 bit(B)，B 标志位表示 TSS 是否正在被使用
  - 随后将 cs 的高 16 位清零， 这是为了防止有一些垃圾字符进入 cs

```
asmlinkage void prepare_fault_handler(unsigned long
target_ip,
      struct tss_struct* cur, struct tss_struct* pre,
struct df_stk* stk)
{
    unsigned int cpu = smp_processor_id();
    clear_busy_flag_in_tss_descriptor(cpu);
    stk->cs &= 0x0000ffff;
    if (pre->x86_tss.cs == __KERNEL_CS && \
        pre->x86_tss.sp ≤ TASK_SIZE) {
        stk->cs = __KU_CS_EXCEPTION;
    }
    pre->x86_tss.ip = target_ip;
    pre->x86_tss.cs = __KERNEL_CS;
    pre->x86_tss.flags &= (~(X86_EFLAGS_TF |
X86_EFLAGS_IF));

    pre->x86_tss.sp = (unsigned long)stk;
    pre->x86_tss.ss = __KERNEL_DS;
```

04 KML 技术内幕 - 异常处理 - task.c: prepare_fault_handler

- 通过 call_helper 执行了 C 语言函数 prepare_fault_handler:
  - sysenter 系统调用发生时设置 stk→cs 字段为之前见到过的 __KU_CS_EXCEPTION 字段
  - 设置 tss 的 ip 为传入的 target_ip， 其指向 double_fault_fixup 汇编函数
  - 设置 tss 的 sp 为 KML 自定义的 stk 结构

```
struct df_stk {
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
};
```

```
asmlinkage void prepare_fault_handler(unsigned long
target_ip,
      struct tss_struct* cur, struct tss_struct* pre,
struct df_stk* stk)
{
    unsigned int cpu = smp_processor_id();
    clear_busy_flag_in_tss_descriptor(cpu);
    stk->cs &= 0x0000ffff;
    if (pre->x86_tss.cs == __KERNEL_CS && \
        pre->x86_tss.sp ≤ TASK_SIZE) {
        stk->cs = __KU_CS_EXCEPTION;
    }
    pre->x86_tss.ip = target_ip;
    pre->x86_tss.cs = __KERNEL_CS;
    pre->x86_tss.flags &= (~(X86_EFLAGS_TF |
X86_EFLAGS_IF));

    pre->x86_tss.sp = (unsigned long)stk;
    pre->x86_tss.ss = __KERNEL_DS;
```

04 KML 技术内幕 - 异常处理 - task.c: entry_32.S:double_fault_fixup

- 现在再看一下在上一步设置 ip 指向的汇编代码 double_fault_fixup:
  - 当执行这段代码时， 其实已经切换到了 TSS 段中， 使用其私有的内核栈
  - 其重建了栈结构， 将 eax，edx，ecx 的内容进行保存；这是由于 eax，edx，ecx 这三个寄存器在 fastcall 编译选项加入时会被使用用来传递参数， 因而在执行 do_interrupt_handling 之前需要保存
  - cr2 寄存器是页故障线性地址寄存器，保存最后一次出现页故障的全 32 位线性地址， 也压入栈
  - 调用真正的内核原生处理代码处理可能发生的中断
  - 弹栈， 清空其中内容

```
ENTRY(double_fault_fixup)
    pushl %eax
    pushl %edx
    pushl %ecx
    movl %cr2, %eax
    pushl %eax

    call do_interrupt_handling

    popl %eax
    movl %eax, %cr2
    popl %ecx
    popl %edx
    popl %eax

    pushl $PAGE_FAULT_ERROR_CODE
    pushl $do_page_fault
    jmp error_code
```

04 KML 技术内幕 - 异常处理 - task.c: entry_32.S:double_fault_fixup

- 现在再看一下在上一步设置 ip 指向的汇编代码 double_fault_fixup:
  - 随后将两个内容压入栈中， 分别是页错误相关的错误码和处理函数
  - 跳转至 error_code 执行内核异常处理
- 关于 error_code 汇编代码， 做一些补充：
  - 在原生内核中， 执行代码前必须要求压入上述两个值
  - 除了 Device not avaliable 异常， 其余异常处理程序都会 jmp 到 error_code 代码段中
  - 其会存储若干寄存器值至栈中， 调用刚刚传入的高级 C 语言函数

```
ENTRY(double_fault_fixup)
    pushl %eax
    pushl %edx
    pushl %ecx
    movl %cr2, %eax
    pushl %eax

    call do_interrupt_handling

    popl %eax
    movl %eax, %cr2
    popl %ecx
    popl %edx
    popl %eax

    pushl $PAGE_FAULT_ERROR_CODE
    pushl $do_page_fault
    jmp error_code
```

04 KML 技术内幕 - 异常处理 - entry_32.S:double_fault_task

- 最后，执行 ret_from_task_without_iret 返回！
  - 由于 iret 会启用 NMI，因而手动返回到 TSS 中

```
.macro ret_from_task_without_iret cur_tss tss_desc
    /* clear NT in EFLAGS */
    pushfl
    andl $~X86_EFLAGS_NT, (%esp)
    popfl

    movl TSS_ESP0(\cur_tss), %esp

    /* We don't use iret, because it will enable
NMI */
    ljmp $(\tss_desc*8), $0x0
.endm
```

```
ENTRY(double_fault_task)
    movl 4(%esp), %edi      # get current TSS.
/* %edi = current_tss */
    movl 8(%esp), %ebx      # get normal TSS.
/* %ebx = prev_tss */
    # get kernel stack.
    kml_get_kernel_stack %ebx, %esi
    movl %esi, %esp
/* From now on, we can use stack. */
    # recreate stack layout as if normal interrupt
    kml_recreate_kernel_stack_layout %ebx
    call_helper prepare_fault_handler,  \
        $double_fault_fixup, %edi, %ebx
    ret_from_task_without_iret %edi, GDT_ENTRY_TSS
    jmp double_fault_task
```

04 KML 技术内幕 - 异常处理 - cpu_init

- 对 tss 的设置和初始化在 cpu_inti 函数中， 总体来说：
  - 该函数首先取了 2 个 per_cpu 变量的指针， 这两个变量都是 tss_struct 类型的 .
  - 随后对这两个变量进行初始化
  - 将 tss 插入到 gdt 中相应位置
- 我们将深入考察一下前两个部分涉及的代码片段
- 对于每 CPU 变量的考察暂时告一段落， 最后再看一下 init_doublefault_tss 中做了什么

```
#ifdef CONFIG_KERNEL_MODE_LINUX
    struct tss_struct* doublefault_tss =
&per_cpu(doublefault_tsses, cpu);
    struct tss_struct* nmi_tss = &per_cpu(nmi_tsses, cpu);
#endif

#ifndef CONFIG_KERNEL_MODE_LINUX
#ifdef CONFIG_DOUBLEFAULT
    /* Set up doublefault TSS pointer in the GDT */
    __set_tss_desc(cpu, GDT_ENTRY_DOUBLEFAULT_TSS,
&doublefault_tss);
#endif
#else
    init_doublefault_tss(cpu);
    init_nmi_tss(cpu);
    __set_tss_desc(cpu, GDT_ENTRY_DOUBLEFAULT_TSS,
doublefault_tss);
    __set_tss_desc(cpu, GDT_ENTRY_NMI_TSS, nmi_tss);
```

04 KML 技术内幕 - 异常处理 - task.c: init_doublefault_tss

- init_doublefault_tss 中：
  - dft_stack 指向的是之前看到过的 KML 自定义的栈结构

```
struct dft_stack_struct {
    unsigned long error_code;
    struct tss_struct* this_tss;
    struct tss_struct* normal_tss;
};
```

  - 这段代码将 doublefault_tss 和这段 TSS 结构做了关联，使得在处理 doublefault 异常时可以使用到该段内的栈结构 .

```
void __cpuinit init_doublefault_tss(int cpu)
{
    struct tss_struct* tss = &per_cpu(init_tss, cpu);
    struct tss_struct* doublefault_tss =
&per_cpu(doublefault_tsses, cpu);
    struct dft_stack_struct* dft_stack =
&per_cpu(dft_stacks, cpu);

    doublefault_tss->x86_tss.sp = (unsigned long)
(&(dft_stack->error_code) + 1);
    doublefault_tss->x86_tss.sp0 = doublefault_tss-
>x86_tss.sp;

    dft_stack->this_tss = doublefault_tss;
    dft_stack->normal_tss = tss;

}
```

04 KML 技术内幕 - 异常处理

- 最后将与 doublefault 问题类似的几个困难略作介绍：
  - Stack switching: 由于 CPU 不会主动为 KML 程序切换运行栈，因而 KML 为了处理异常需要手动负责栈的切换．目前的 KML 使用了 GDT 中的 TSS 段（如前所述），因而可以通过只使用一个通用寄存器的情况下获得内核栈的地址
  - Interrupt lost: 考虑一个运行在内核态的用户程序，但其 esp 寄存器指向了一段尚未被映射到其地址空间的内存．此时如果发生了外界的中断，CPU 首先将 ACK 该中断请求，随后其将尝试中断当前运行的用户程序，但是其又因为没有一个可用的栈来保存执行上下文，因此 CPU 产生 double fault 异常，这将由 KML 的 task 机制 (Stack Stavation) 进行处理．问题在于 double fault 的处理者只知道有关用户程序的信息而完全不知道中断到达的信息，这就导致其将恢复用户程序并不处理该中断请求... 中断控制器也会认为 CPU 不会服务该中断因而此后也不会在发送该中断请求，这就导致了中断的丢失．

04 KML 技术内幕 - 异常处理

- 最后将与 doublefault 问题类似的几个困难略作介绍：
  - Non-maskable interrupt: 如果在可屏蔽中断发生后、内存栈从用户栈切换到内核栈之间， 不可屏蔽中断发生了， 其产生了一个 page fault 异常， 那么 doublefault 处理程序也会主动处理之前的可屏蔽中断因为其还未被解决． 但当它完成后返回到该可屏蔽中断的处理例程中， 导致该处理例程会尝试取处理一个已经解决的中断请求．

04 KML 技术内幕 - 异常处理

- KML 技术中的内核态执行用户程序、中断 / 异常处理等各类功能涉及到的内核模块较多， 且部分修改代码为汇编指令较难读懂，需要对 linux 内核处理中各个部分有较为深刻的理解（内存管理、中断异常、系统调用、内核同步、进程、信号等）
- 由于本人水平有限， 因而在对 KML 技术具体的代码进行分析时十分不全面也必定存在不少谬误， 只能管中窥豹将其中部分内容进行交流分享！

谢 谢

欢迎交流合作

2023 年 3 月 22 日