

# Linux Developer Conference Brazil 2019

## Linux kernel debugging: going beyond printk messages



Embedded Labworks



# SOBRE ESTE DOCUMENTO

- x Este documento é disponibilizado sob a Licença Creative Commons BY-SA 3.0.

<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

- x Os fontes deste documento estão disponíveis em:

<https://sergioprado.org/palestras/linuxdevbr2019>



 **creative  
commons**





# SOBRE O PALESTRANTE

- x Mais de 20 anos de experiência em desenvolvimento de software para sistemas embarcados.
- x Sócio da Embedded Labworks, onde atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados.  
<https://e-labworks.com>
- x Ativo na comunidade de sistemas embarcados no Brasil, sendo um dos criadores do site Embarcados, administrador do grupo sis\_embarcados no Google Groups, além de manter um blog pessoal sobre assuntos da área.  
<https://sergioprado.org>
- x Colaborador de alguns projetos de software livre, incluindo o Buildroot e o kernel Linux.





# DEPURAÇÃO PASSO-A-PASSO

1. Entender o problema.
2. Reproduzir o problema.
3. Identificar a origem do problema.
4. Corrigir o problema.
5. Resolveu? Se sim, comemore! Se não, volte para o passo 1.





# TIPOS DE PROBLEMAS

- x Podemos considerar como os 5 principais tipos de problemas em software:
  - x Crash.
  - x Travamento.
  - x Lógica/implementação.
  - x Vazamento de recurso.
  - x Performance.










# FERRAMENTAS E TÉCNICAS

- x Para resolver estes problemas, estas são as 5 principais técnicas e ferramentas de depuração que podemos utilizar:
  - x Conhecimento.
  - x Logs.
  - x Tracing.
  - x Depuração interativa.
  - x Frameworks de depuração.





# PROBLEMAS VS TÉCNICAS

	Crash	Travamento	Lógica	Vazamento de recurso	Performance
printf()					





# PROBLEMAS VS TÉCNICAS

	Crash	Travamento	Lógica	Vazamento de recurso	Performance
Conhecimento					
Logs					
Tracing					
Depuração interativa					
Frameworks de depuração					







# Linux Developer Conference Brazil 2019

Logs



# LOGS DO KERNEL

- \* A função `printk()`, definida em `<linux/printk.h>`, é a responsável por imprimir mensagens no kernel, podendo ser chamada tanto em contexto de processo quanto em contexto de interrupção.

```
int printk(const char *s, ...);
```

- \* As mensagens do kernel são armazenadas em um buffer circular, cujo tamanho pode ser definido em tempo de compilação (`CONFIG_LOG_BUF_SHIFT`) ou em tempo de execução (parâmetro de boot `log_buf_len`).
- \* Por padrão, as mensagens de log do kernel são exibidas na console e podem ser emitidas a qualquer momento com a ferramenta `dmesg`.





# DMESG

```
# dmesg
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.18.9 (labworks@ubuntu) (gcc version 7.3.0 ...)
[    0.000000] Linux debugging training: Linux rocks!
[    0.000000] CPU: ARMv7 Processor [412fc09a] revision 10 (ARMv7), cr=10c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction
[    0.000000] OF: fdt: Machine model: Toradex Colibri iMX6DL/S on Viola Carrier
[    0.000000] Memory policy: Data cache writealloc
[    0.000000] cma: Reserved 64 MiB at 0x2c000000
[    0.000000] On node 0 totalpages: 131072
[    0.000000]   Normal zone: 1024 pages used for memmap
[    0.000000]   Normal zone: 0 pages reserved
[    0.000000]   Normal zone: 131072 pages, LIFO batch:31
[    0.000000] random: get_random_bytes called from start_kernel+0xa0/0x4a4 with
[....]
[    0.511728] clocksource: Switched to clocksource mxc_timer1
[    0.934679] VFS: Disk quotas dquot_6.6.0
[    0.935103] VFS: Dquot-cache hash table entries: 1024 (order 0, 4096 bytes)
[    0.981523] NET: Registered protocol family 2
[    0.983953] tcp_listen_portaddr_hash hash table entries: 512 (order: 0, 6144)
[    0.984014] TCP established hash table entries: 4096 (order: 2, 16384 bytes)
[....]
```





# NÍVEIS DE LOG

- x As mensagens de log do kernel possuem níveis de prioridade.

```
printk(KERN_WARNING "warning: skipping physical page 0\n");
```

- x Os seguintes níveis de prioridade são definidos pelo kernel:

0 (KERN_EMERG)	system is unusable
1 (KERN_ALERT)	action must be taken immediately
2 (KERN_CRIT)	critical conditions
3 (KERN_ERR)	error conditions
4 (KERN_WARNING)	warning conditions
5 (KERN_NOTICE)	normal but significant condition
6 (KERN_INFO)	informational
7 (KERN_DEBUG)	debug-level messages





# MENSAGENS DE DEBUG

- x Por padrão, as mensagens de nível debug não são compiladas.
- x Para que as mensagens de debug sejam compiladas, é necessário definir a macro DEBUG. Isso pode ser feito no Makefile do arquivo que se deseja habilitar as mensagens de debug.

```
CFLAGS_[filename].o := -DDEBUG
```

- x Por exemplo, para compilar as mensagens de debug do arquivo drivers/char/raw.c, é necessário inserir a linha abaixo no arquivo drivers/char/Makefile:

```
CFLAGS_raw.o := -DDEBUG
```





# DEBUG DINÂMICO

- x Outra forma de habilitar as mensagens de debug é através da funcionalidade de debug dinâmico (`CONFIG_DYNAMIC_DEBUG`).
- x Esta funcionalidade permite filtrar as mensagens de debug por arquivo, função ou até por linha de código-fonte!
- x As mensagens de debug podem ser habilitadas individualmente através de um arquivo de controle disponibilizado via `debugfs`.
- x A documentação desta funcionalidade está disponível no código-fonte do kernel em `Documentation/admin-guide/dynamic-debug-howto.rst`.





# KERNEL OOPS

- x **Kernel oops** é um mecanismo de comunicação do kernel para notificar o usuário de que algum erro aconteceu.
- x Este erro pode acontecer por diversos motivos, como por exemplo acesso ilegal à memória ou execução de instruções inválidas.
- x Durante o oops, o kernel emite uma mensagem nos logs com o status atual do sistema no momento em que o problema aconteceu, incluindo um dump dos registradores e o backtrace do stack.





# KERNEL PANIC

- x Após o oops, o kernel irá tentar se recuperar e resumir a execução, mas dependendo do erro, nem sempre isso é possível.
- x Nestes casos, o kernel pode travar após o oops com um **kernel panic**.
- x Em um kernel panic, a execução do sistema operacional é interrompida e uma mensagem com o motivo do panic é exibida nos logs do kernel.







# EXEMPLO KERNEL PANIC

```
[ 3500.297857] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[ 3500.306083] pgd = da546965
[ 3500.308813] [00000000] *pgd=293ab831, *pte=00000000, *ppte=00000000
[ 3500.315208] Internal error: Oops: 817 [#1] SMP ARM
[ 3500.320009] Modules linked in:
[ 3500.323077] CPU: 0 PID: 176 Comm: sh Not tainted 4.18.9 #20
[ 3500.328651] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[ 3500.335192] PC is at sysrq_handle_crash+0x2c/0x38
[ 3500.339907] LR is at arm_heavy_mb+0x28/0x48
[ 3500.344095] pc : [<c03f69f8>]   lr : [<c00243e4>]   psr: 60060013
[ 3500.350363] sp : d938be30   ip : d938be20   fp : d938be44
[ 3500.355589] r10: 00000002   r9 : 0131f6f0   r8 : c0b77ec4
[ 3500.360816] r7 : 00000000   r6 : 00000007   r5 : 00000001   r4 : 00000000
[ 3500.367346] r3 : 00000000   r2 : 00000000   r1 : 00000730   r0 : 00000063
[ 3500.373877] Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
[ 3500.381014] Control: 10c5387d  Table: 2977804a  DAC: 00000051
[...]
```

[ 3500.519440] [<c03f69f8>] (sysrq\_handle\_crash) from [<c03f7000>] (\_\_handle\_sysrq+0x9c/0x17c)

[ 3500.527802] [<c03f7000>] (\_\_handle\_sysrq) from [<c03f75ec>] (write\_sysrq\_trigger+0x68/0x78)

[ 3500.536163] [<c03f75ec>] (write\_sysrq\_trigger) from [<c01f7f44>] (proc\_reg\_write+0x6c/0x94)

[ 3500.544522] [<c01f7f44>] (proc\_reg\_write) from [<c0189b80>] (\_\_vfs\_write+0x44/0x160)

[ 3500.552271] [<c0189b80>] (\_\_vfs\_write) from [<c0189e2c>] (vfs\_write+0xb0/0x178)

[ 3500.559584] [<c0189e2c>] (vfs\_write) from [<c018a068>] (ksys\_write+0x58/0xbc)

[ 3500.566725] [<c018a068>] (ksys\_write) from [<c018a0e4>] (sys\_write+0x18/0x1c)

[ 3500.573866] [<c018a0e4>] (sys\_write) from [<c0009000>] (ret\_fast\_syscall+0x0/0x28)

[...]

[ 3500.618692] Kernel panic - not syncing: Fatal exception





# ADDR2LINE

- x A ferramenta `addr2line` é capaz de converter um endereço de memória em uma linha de código-fonte:

```
$ arm-linux-addr2line -f -e vmlinux 0xc03f69f8  
sysrq_handle_crash  
/opt/labs/ex/linux/drivers/tty/sysrq.c:147
```





# GDB DISASSEMBLE

```
$ arm-linux-gdb vmlinux
(gdb) disassemble /m sysrq_handle_crash
Dump of assembler code for function sysrq_handle_crash:
136 {
    0xc03f69cc <+0>: mov r12, sp
    0xc03f69d0 <+4>: push    {r4, r5, r11, r12, lr, pc}
    0xc03f69d4 <+8>: sub r11, r12, #4
    0xc03f69d8 <+12>:  push    {lr}          ; (str lr, [sp, #-4]!)
    0xc03f69dc <+16>:  bl  0xc001b420 <__gnu_mcount_nc>

137     char *killer = NULL;
[... ]
146     wmb();
    0xc03f69e8 <+28>:  mov r4, #0
    0xc03f69f0 <+36>:  mcr 15, 0, r4, cr7, cr10, {4}
    0xc03f69f4 <+40>:  bl  0xc00243bc <arm_heavy_mb>

147     *killer = 1;
    0xc03f69f8 <+44>:  strb    r5, [r4]

148 }
[... ]
```





# GDB INFO/LIST

```
$ arm-linux-gdb vmlinux
```

```
(gdb) info line *(sysrq_handle_crash+0x2c)
```

Line 147 of "drivers/tty/sysrq.c" starts at address 0xc03f69f8 <sysrq\_handle\_crash+44> and ends at 0xc03f69fc <sysrq\_handle\_crash+48>.

```
(gdb) list *(sysrq_handle_crash+0x2c)
```

0xc03f69f8 is in sysrq\_handle\_crash (drivers/tty/sysrq.c:147).

```
142      * complaint from the kernel before the panic.
```

```
143      */
```

```
144      rcu_read_unlock();
```

```
145      panic_on_oops = 1; /* force panic */
```

```
146      wmb();
```

```
147      *killer = 1;
```

```
148 }
```

```
149 static struct sysrq_key_op sysrq_crash_op = {
```

```
150     .handler      = sysrq_handle_crash,
```

```
151     .help_msg     = "crash(c)",
```





# Linux Developer Conference Brazil 2019

Tracing



# TRACING NO KERNEL LINUX

- x Existem dois principais tipos de implementação de tracing no kernel Linux: tracing estático e tracing dinâmico.
- x Tracing estático é implementado através de probes estáticos presentes no código-fonte. Possuem baixa carga de processamento, porém o código rastreado é limitado e definido em tempo de compilação.
- x Tracing dinâmico é implementado através probe dinâmicos injetados no código normalmente através de interrupção de software, permitindo definir em tempo de execução o código a ser rastreado. Possui uma certa carga de processamento, mas a abrangência do código-fonte a ser rastreado é bem maior.
- x A documentação sobre tracing no kernel Linux está disponível no código-fonte do kernel em `Documentation/trace/`.





# PROBES ESTÁTICOS

```
static int gpiod_get_value(const struct gpio_desc *desc)
{
    struct gpio_chip      *chip;
    int value;
    int offset;

    if (!desc)
        return 0;
    chip = desc->chip;
    offset = gpio_chip_hwgpio(desc);
    /* Should be using gpio_get_value_cansleep() */
    WARN_ON(chip->can_sleep);
    value = chip->get ? chip->get(chip, offset) : 0;
    trace_gpio_value(desc_to_gpio(desc), 1, value);
    return value;
}
```





# PROBES DINÂMICOS

```
void input_set_abs_params(struct input_dev *dev, unsigned int axis,  
                        int min, int max, int fuzz, int flat)  
{  
    struct input_absinfo *absinfo;  
  
    input_alloc_absinfo(dev);  
    if (!dev->absinfo)  
        return;  
  
    absinfo = &dev->absinfo[axis];  
    absinfo->minimum = min;  
    absinfo->maximum = max;  
    absinfo->fuzz = fuzz;  
    absinfo->flat = flat;  
  
    dev->absbit[BIT_WORD(axis)] |= BIT_MASK(axis);  
}
```

Software INT

Save context

Probe function

Restore context







# FRAMEWORKS E FERRAMENTAS

- x Diversos frameworks e ferramentas utilizam estes recursos de rastreamento para instrumentar o kernel, incluindo:
  - x Ftrace.
  - x Trace-cmd.
  - x Kernelshark.
  - x SystemTap.
  - x Perf.
  - x BCC (BPF Compiler Collection).





# FTRACE

- x O ftrace é um dos principais frameworks de tracing do kernel Linux, e pode ser utilizado tanto para depuração e identificação de problemas quanto para análise de performance e latência.
- x Utiliza mecanismos de tracing estático e dinâmico do kernel.
- x Armazena as informações de tracing em um buffer circular em memória.
- x Interface com o usuário através de arquivos via sistema de arquivos virtual tracefs.





# FTRACE NO KERNEL

- x É habilitado no kernel através da opção `CONFIG_FTRACE`.
- x Diversos tracers estão disponíveis, incluindo:
  - x **Function Tracer**: tracing de chamada de função do kernel.
  - x **Kernel Function Profiler**: estatísticas de chamada de função do kernel.
  - x **Interrupts-off Latency Tracer**: tempo gasto com as interrupções desabilitadas.
  - x **Preemption-off Latency Tracer**: tempo gasto com a preempção do kernel desabilitada.
  - x **Scheduling Latency Tracer**: tempo máximo de latência para escalonar uma tarefa de alta prioridade.





# FTRACE NO KERNEL (cont.)

```
Terminal
File Edit View Search Terminal Help
.config - Linux/arm 4.18.9 Kernel Configuration
> Search (CONFIG_FTRACE) > Kernel hacking > Tracers

Tracers
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
excluded <M> module < > module capable

-- Tracers
-* Kernel Function Tracer
[*] Kernel Function Graph Tracer
[*] Enable trace events for preempt and irq disable/enable
[*] Interrupts-off Latency Tracer
[*] Scheduling Latency Tracer
[*] Tracer to detect hardware latencies (like SMIs)
[*] Trace syscalls
-* Create a snapshot trace buffer
-* Allow snapshot to swap per CPU
Branch Profiling (No branch profiling) --->
[*] Trace max stack
[*] Support for tracing block IO actions
[*] Enable uprobes-based dynamic events
[*] enable/disable function tracing dynamically
[*] Kernel function profiler
[ ] Perform a startup test on ftrace
[ ] Add tracepoint that benchmarks tracepoints
< > Ring buffer benchmark stress tester
[ ] Ring buffer startup self test
[ ] Show eval mappings for trace events
[*] Trace gpio events

<Select> < Exit > < Help > < Save > < Load >
```





# USANDO O FTRACE

```
# mount -t tracefs none /sys/kernel/tracing
```

```
# cd /sys/kernel/tracing/
```

```
# cat available_tracers
```

hwlat	blk	function_graph	wakeup_dl	wakeup_rt
wakeup	irqsoff	function	nop	





## USANDO FTRACE (cont.)

- x O usuário pode selecionar um tracer escrevendo seu nome no arquivo `current_tracer`.

```
# echo function > current_tracer
```

- x O buffer de tracing pode ser obtido lendo os arquivos `trace` ou `trace_pipe`.

```
# cat trace
```

```
# cat trace_pipe
```





# FUNCTION TRACER

```
# echo function > current_tracer
```

```
# cat trace
```

```
# tracer: function
```

```
#
```

```
#
```

```
#
```

```
#
```

```
#
```

```
#
```

```
#
```

```
#
```

```

#          _-----=> irqsoff
#         /_-----=> need-resched
#        |/_-----=> hardirq/softirq
#       ||/_-----=> preempt-depth
#      |||/_-----=> delay
#     ||||
# TASK-PID  CPU#  ||||  TIMESTAMP  FUNCTION
#   |   |   |   |   |   |
<idle>-0   [001] d...  23.695208: _raw_spin_lock_irqsave <-hrtimer_next_event_wi...
<idle>-0   [001] d...  23.695209: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0   [001] d...  23.695210: __next_base <-__hrtimer_next_event_base
<idle>-0   [001] d...  23.695211: __hrtimer_next_event_base <-hrtimer_next_event...
<idle>-0   [001] d...  23.695212: __next_base <-__hrtimer_next_event_base
<idle>-0   [001] d...  23.695213: __next_base <-__hrtimer_next_event_base
<idle>-0   [001] d...  23.695214: _raw_spin_unlock_irqrestore <-hrtimer_next_eve...
<idle>-0   [001] d...  23.695215: get_iowait_load <-menu_select
<idle>-0   [001] d...  23.695217: tick_nohz_tick_stopped <-menu_select
<idle>-0   [001] d...  23.695218: tick_nohz_idle_stop_tick <-do_idle
<idle>-0   [001] d...  23.695219: rcu_idle_enter <-do_idle
<idle>-0   [001] d...  23.695220: call_cpuidle <-do_idle
<idle>-0   [001] d...  23.695221: cpuidle_enter <-call_cpuidle
```

```
[...]
```





# TRACE-CMD & KERNELSHARK

- x Trace-cmd é uma ferramenta de linha de comando que serve de interface para o ftrace.
- x É capaz de configurar o ftrace, ler o buffer e salvar os dados em um arquivo (trace.dat) para posterior análise.
- x Kernelshark é uma ferramenta gráfica que serve de frontend para o arquivo trace.dat gerado pela ferramenta trace-cmd.

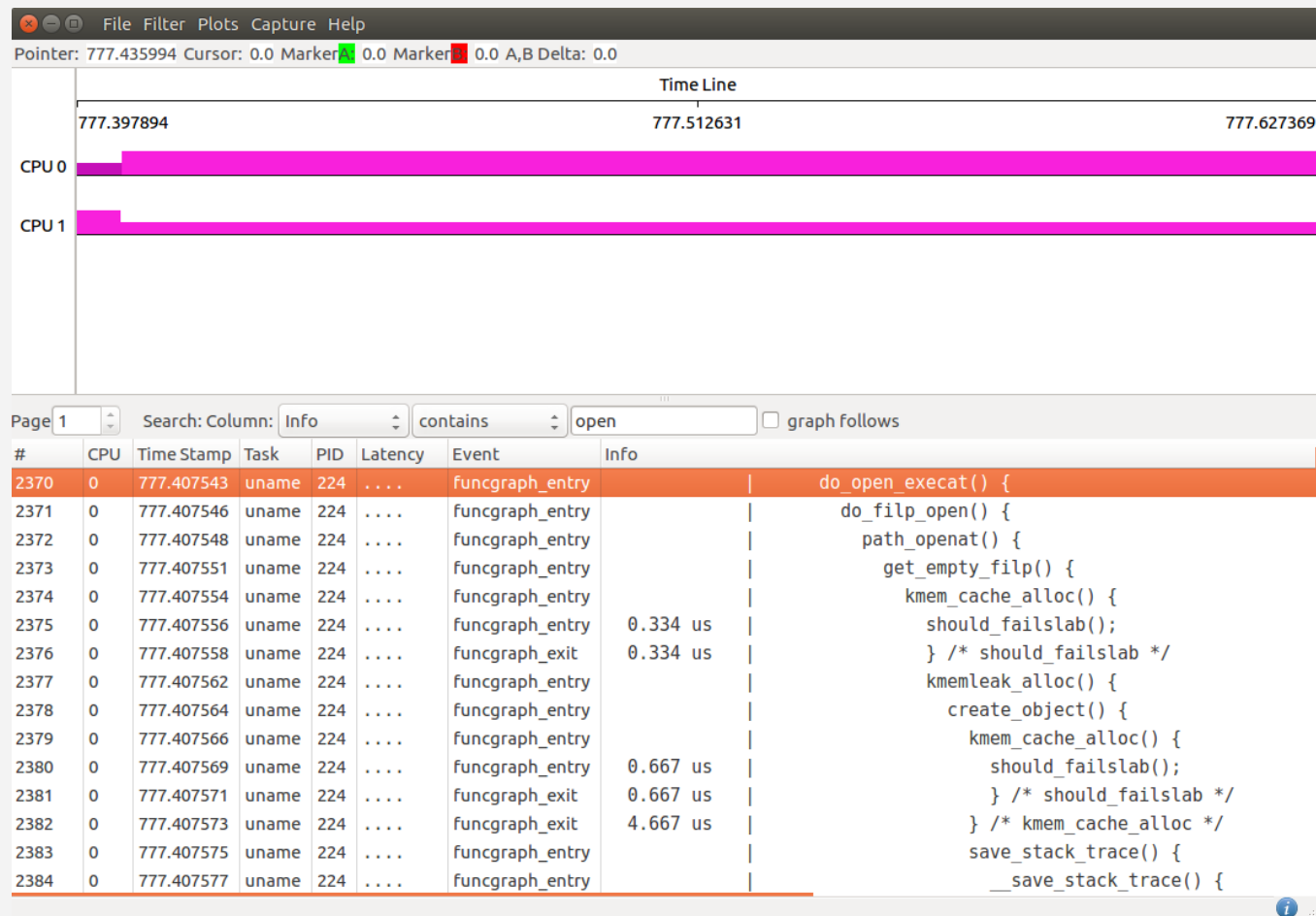






# KERNELSHARK

```
$ kernelshark trace.dat
```





# Linux Developer Conference Brazil 2019

Depuração interativa



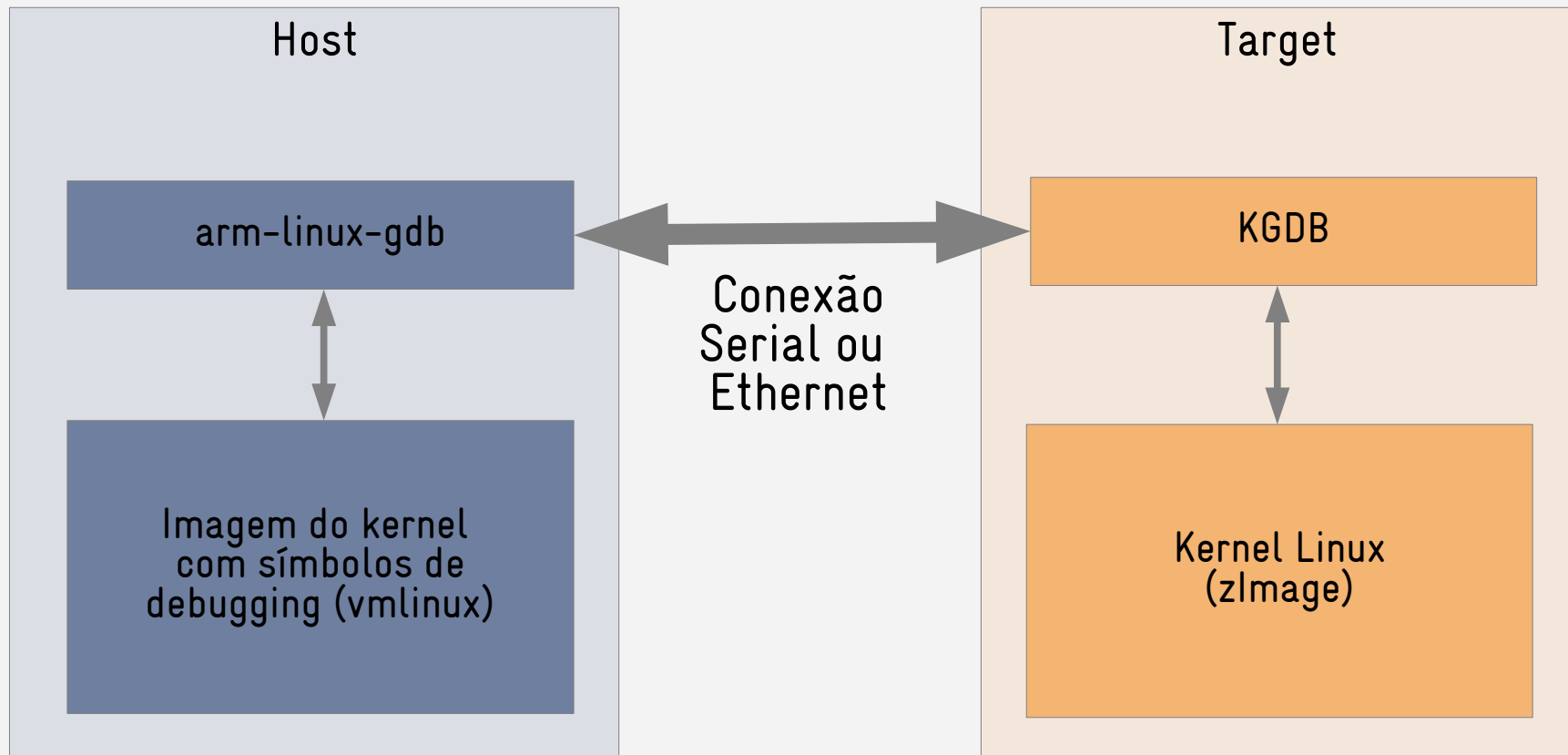
# DEPURANDO O KERNEL COM O GDB

- x **Problema 1:** Como utilizar o kernel para depurar ele mesmo?
- x **Problema 2:** o código-fonte e as ferramentas de desenvolvimento estão no host e a imagem do kernel está em execução no target.
- x **Solução:** arquitetura cliente/servidor. O kernel Linux possui uma implementação de servidor do GDB chamada KGDB que se comunica com o cliente do GDB pela rede ou porta serial.





# DEPURANDO O KERNEL COM O GDB (cont.)





# KGDB

- × O KGDB é uma implementação de servidor GDB integrada ao kernel Linux.  
<https://www.kernel.org/doc/html/latest/dev-tools/kgdb.html>
- × Suporta comunicação via porta serial (disponível no kernel mainline) e pela rede (necessário aplicar patch).
- × Disponível no kernel mainline desde a versão 2.6.26 (x86 e sparc) e 2.6.27 (arm, mips e ppc).
- × Possibilita controle total sobre a execução do kernel no target, incluindo leitura e escrita em memória, execução passo-a-passo e até breakpoints em rotinas de tratamento de interrupção!





# DEPURANDO O KERNEL COM O KGDB

- x São três os passos para depurar o kernel Linux com o KGDB:
  1. Compilar o kernel com suporte à depuração via KGDB.
  2. Configurar e colocar o target no modo de depuração do KGDB.
  3. Configurar o toolchain no host para se comunicar com KGDB no target via serial ou rede.





# 1. CONFIGURANDO O KERNEL

- x Para usar o KGDB, é necessário habilitar e recompilar o kernel Linux com as seguintes opções:
  - x CONFIG\_KGDB: habilita o KGDB.
  - x CONFIG\_KGDB\_SERIAL\_CONSOLE: habilita o driver de I/O para a comunicação entre o host e o target pela porta serial.
  - x CONFIG\_MAGIC\_SYSRQ: habilita a funcionalidade de magic sysrq key para colocar o kernel em modo debug.
  - x CONFIG\_DEBUG\_INFO: compila o kernel com símbolos de debugging.
  - x CONFIG\_FRAME\_POINTER: ajuda a produzir backtraces do stack mais confiáveis.





## 2. CONFIGURANDO O TARGET

- × O target pode ser configurado para entrar no modo KGDB em tempo de boot através da linha de comandos do kernel ou em tempo de execução através de arquivos disponíveis no /proc.

- × Para configurar o KGDB em tempo de boot, utilize os parâmetros de boot `kgdboc` e `kgdbwait`, conforme exemplo abaixo:

```
kgdboc=ttymxc0,115200 kgdbwait
```

- × Em tempo de execução, podemos utilizar os comandos abaixo para colocar o kernel em modo debug:

```
# echo ttymxc0 > /sys/module/kgdboc/parameters/kgdboc
```

```
# echo g > /proc/sysrq-trigger
```







### 3. INICIANDO A DEPURAÇÃO NO HOST (A)

- x No host, execute o cliente do GDB passando a imagem ELF do kernel com símbolos de debugging:

```
$ arm-linux-gdb vmlinux -tui
```

- x Na linha de comandos do GDB, configure a porta serial e inicie a conexão com o KGDB:

```
(gdb) set serial baud 115200
```

```
(gdb) target remote /dev/ttyUSB0
```

- x Se o KGDB estiver em execução no target, o processo de depuração do kernel deverá ser iniciado automaticamente.





# AGENT PROXY

- x Caso você esteja usando a serial tanto para a console quanto para a depuração via KGDB, é necessário o uso de um proxy para gerenciar a comunicação pela serial.
- x Um proxy bem simples e funcional está disponível nos repositórios do kernel Linux.

```
$ git clone https://kernel.googlesource.com/pub/scm/utils/kernel/kgdb/agent-proxy
```

```
$ cd agent-proxy/
```

```
$ make
```





### 3. INICIANDO A DEPURACÃO NO HOST (B)

- × Para iniciar a depuração pela porta serial utilizando o proxy, primeiro execute o programa do proxy:

```
$ ./agent-proxy 5550^5551 0 /dev/ttyUSB0,115200
```

- × Abra um terminal e execute o comando telnet para iniciar a conexão com a console do target:

```
$ telnet localhost 5550
```

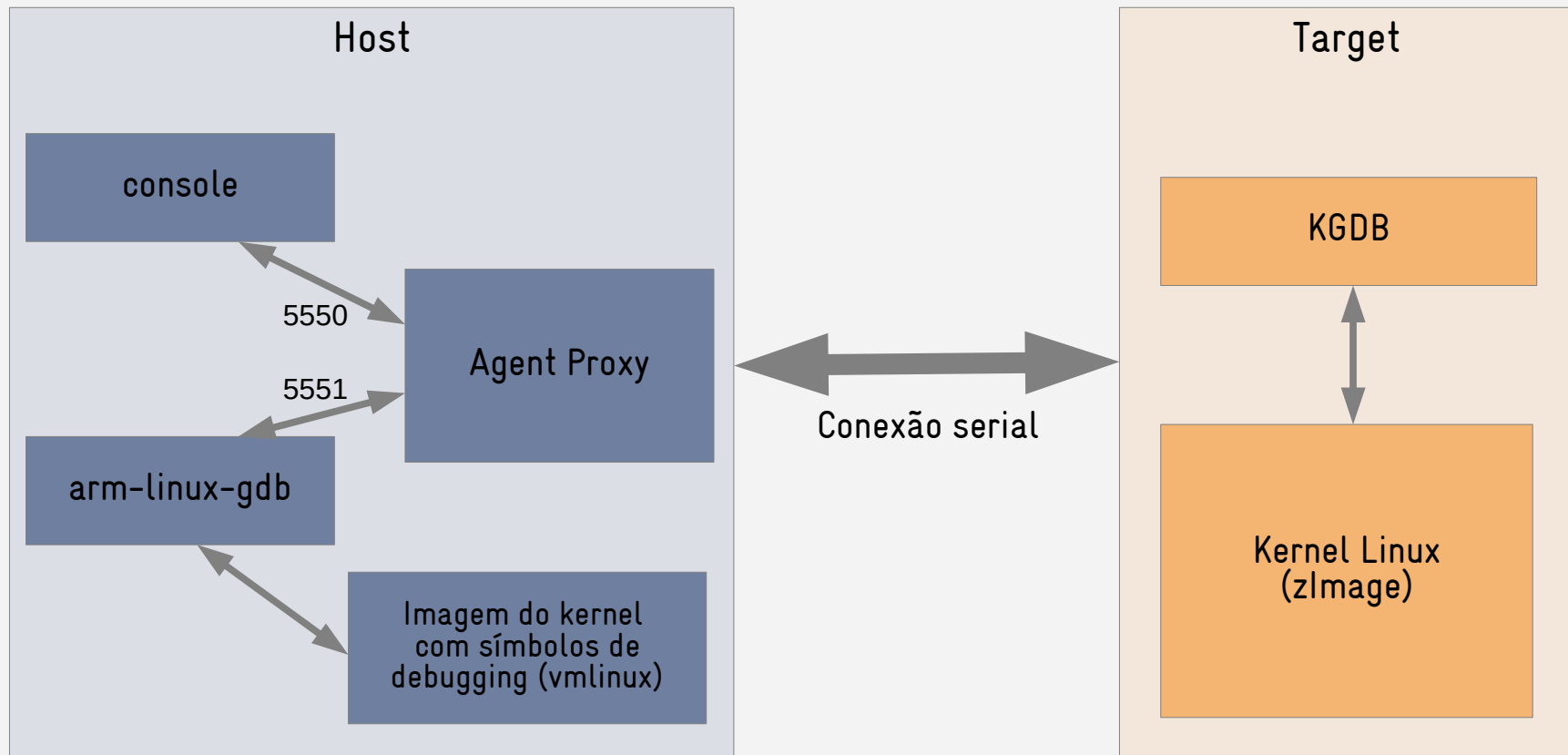
- × Em um outro terminal, se conecte ao KGDB utilizando o GDB do toolchain de compilação cruzada:

```
$ arm-linux-gdb vmlinux -tui  
(gdb) target remote localhost:5551
```





## AGENT PROXY (cont.)





# Linux Developer Conference Brazil 2019

Frameworks de depuração



# KERNEL HACKING

```
Terminal File Edit View Search Terminal Help
.config - Linux/arm 4.18.9 Kernel Configuration
> Kernel hacking

Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----).
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
excluded <M> module < > module capable

printk and dmesg options --->
  Compile-time checks and compiler options --->
  *- Magic SysRq key
  (0x1) Enable magic SysRq key functions by default
  [*] Enable magic SysRq key over serial
  *- Kernel debugging
  Memory Debugging --->
  [ ] Code coverage for fuzzing
  [ ] Debug shared IRQ handlers
  Debug Lockups and Hangs --->
  [ ] Panic on Oops
  (5) panic timeout
  [ ] Collect scheduler debugging info
  [ ] Collect scheduler statistics
  [ ] Detect stack corruption on calls to schedule()
  [ ] Enable extra timekeeping sanity checking
  [*] Debug preemptible kernel
  Lock Debugging (spinlocks, mutexes, etc...) --->
  *- Stack backtrace support
  [ ] Warn for all uses of unseeded randomness
  [ ] kobject debugging
  [ ] Verbose BUG() reporting (adds 70K)
  [ ] Debug linked list manipulation
  [ ] Debug priority linked list manipulation
  [ ] Debug SG table operations
  [ ] Debug notifier call chains
  [ ] Debug credential management
  RCU Debugging --->
  [ ] Force round-robin CPU selection for unbound work items
  [ ] Force extended block device numbers and spread them
  (+)

<Select>  < Exit >  < Help >  < Save >  < Load >
```





# VAZAMENTO DE MEMÓRIA

- x Um consumo excessivo de memória do sistema pode estar associado a um problema de vazamento de memória em espaço de kernel.
- x O kernel possui uma funcionalidade chamada `kmemleak`, capaz de monitorar as rotinas de alocação de memória do kernel e identificar possíveis vazamentos de memória.
- x Esta funcionalidade pode ser habilitada através da opção de configuração `CONFIG_DEBUG_KMEMLEAK`.





# VAZAMENTO DE MEMÓRIA

- Com o `kmemleak` habilitado, uma thread do kernel irá monitorar a memória a cada 10 minutos e registrar possíveis regiões de memória alocadas e não liberadas.

```
# ps | grep kmemleak
```

```
root      151      2      0      0      800df728 00000000 S kmemleak
```

- Informações sobre possíveis vazamentos de memória estarão disponíveis em um arquivo no debugfs:

```
# cat /sys/kernel/debug/kmemleak
```







## VAZAMENTO DE MEMÓRIA (cont.)

- Podemos forçar a checagem de memória e criar uma lista de possíveis vazamentos de memória escrevendo scan neste arquivo:

```
# echo scan > /sys/kernel/debug/kmemleak
```

- Para limpar a lista atual de possíveis vazamentos de memória, podemos escrever clear neste arquivo:

```
# echo clear > /sys/kernel/debug/kmemleak
```

- A documentação desta funcionalidade está disponível no código-fonte do kernel em `Documentation/dev-tools/kmemleak.rst`.





# USANDO O KMEMLEAK

- x Primeiro limpe qualquer possível vazamento de memória identificado pelo kernel, executando os comandos abaixo até que nenhum vazamento de memória seja exibido:

```
# echo clear > /sys/kernel/debug/kmemleak  
# echo scan > /sys/kernel/debug/kmemleak
```

- x Depois faça testes no kernel para tentar reproduzir o vazamento de memória.
- x Por fim, execute o scan para verificar se houve alguma ocorrência de vazamento de memória:

```
# echo scan > /sys/kernel/debug/kmemleak  
kmemleak: 6 new suspected memory leaks  
(see /sys/kernel/debug/kmemleak)
```





## USANDO 0 KMEMLEAK (cont.)

```
# cat /sys/kernel/debug/kmemleak
unreferenced object 0xd9868000 (size 30720):
  comm "sh", pid 179, jiffies 4294943731 (age 19.720s)
  hex dump (first 32 bytes):
    00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
    0a 00 07 41 00 00 00 00 00 00 00 00 00 00 28 6e bf d8 ...A.....(n..
  backtrace:
    [] kmalloc_order+0x54/0x5c
    [] kmalloc_order_trace+0x2c/0x10c
    [] gpiod_set_value_cansleep+0x3c/0x54
    [] value_store+0x98/0xd8
    [] dev_attr_store+0x28/0x34
    [] sysfs_kf_write+0x48/0x54
    [] kernfs_fop_write+0xfc/0x1e0
    [] __vfs_write+0x44/0x160
    [] vfs_write+0xb0/0x178
    [] ksys_write+0x58/0xbc
    [] sys_write+0x18/0x1c
    [] ret_fast_syscall+0x0/0x28
    [] 0xbe829888
```





## USANDO O KMEMLEAK (cont.)

```
$ arm-linux-addr2line -f -e vmlinux 0xc03c39ec  
gpiod_set_value_cansleep  
/opt/labs/ex/linux/drivers/gpio/gpiolib.c:3465
```

```
$ arm-linux-gdb vmlinux  
(gdb) list *(gpiod_set_value_cansleep+0x3c)  
0xc03c39ec is in gpiod_set_value_cansleep (drivers/gpio/gpiolib.c:3465).  
3460     void gpiod_set_value_cansleep(struct gpio_desc *desc, int value)  
3461     {  
3462         might_sleep_if(extra_checks);  
3463         VALIDATE_DESC_VOID(desc);  
3464         kmalloc(1024*30, GFP_KERNEL);  
3465         gpiod_set_value_nocheck(desc, value);  
3466     }  
3467     EXPORT_SYMBOL_GPL(gpiod_set_value_cansleep);
```





# TRAVAMENTOS

- x O kernel possui algumas opções para identificação de travamentos em espaço de kernel no menu de configuração "Kernel Hacking".
- x A opção `CONFIG_HARDLOCKUP_DETECTOR` irá identificar problemas de travamento em espaço de kernel por mais de 10 segundos sem deixar uma interrupção executar.
  - x A opção `CONFIG_BOOTPARAM_HARDLOCKUP_PANIC` irá causar um kernel panic em "hard lockups".
- x A opção `CONFIG_SOFTLOCKUP_DETECTOR` irá identificar problemas de travamento em espaço de kernel por mais de 20 segundos sem deixar outras tarefas executarem.
  - x A opção `CONFIG_BOOTPARAM_SOFTLOCKUP_PANIC` irá causar um kernel panic em "soft lockups".





## TRAVAMENTOS (cont.)

- x A opção `CONFIG_DETECT_HUNG_TASK` irá identificar tarefas que travam no estado “Uninterruptible” indefinitivamente.
  - x A opção `CONFIG_BOOTPARAM_HUNG_TASK_PANIC` irá causar um kernel panic em “hung tasks”.
- x Ao identificar um travamento, o kernel imprime nos logs um backtrace do stack da tarefa que travou.





# TRAVAMENTOS (cont.)

```
# hwclock -w -f /dev/rtc1
[ 48.041337] watchdog: BUG: soft lockup - CPU#1 stuck for 22s! [hwclock:180]
[ 48.048322] Modules linked in:
[ 48.051396] CPU: 1 PID: 180 Comm: hwclock Not tainted 4.18.9 #51
[ 48.057412] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[ 48.063964] PC is at snvs_rtc_set_time+0x60/0xc8
[ 48.068599] LR is at _raw_spin_unlock_irqrestore+0x40/0x54
[ 48.074093] pc : [<c0516eec>] lr : [<c0723aa8>] psr: 60060013
[ 48.080367] sp : d949fdf8 ip : d949fd78 fp : d949fe2c
[ 48.085599] r10: c0786554 r9 : bef2bc94 r8 : 00000000
[ 48.090832] r7 : d8e71450 r6 : c0bc74a0 r5 : d840b410 r4 : d949fe58
[ 48.097368] r3 : 1e6a8abe r2 : 1e6a8abe r1 : 00000000 r0 : 00000000
[ 48.103904] Flags: nZCv IRQs on FIQs on Mode SVC_32 ISA ARM Segment none
[ 48.111047] Control: 10c5387d Table: 2980804a DAC: 00000051
[ 48.116805] CPU: 1 PID: 180 Comm: hwclock Not tainted 4.18.9 #51
[ 48.122818] Hardware name: Freescale i.MX6 Quad/DualLite (Device Tree)
[... ]
[ 48.253808] [<c0009a30>] (__irq_svc) from [<c0516eec>] (snvs_rtc_set_time+0x60/0xc8)
[ 48.261571] [<c0516eec>] (snvs_rtc_set_time) from [<c050c358>] (rtc_set_time+0x94/0x1f0)
[ 48.269676] [<c050c358>] (rtc_set_time) from [<c050dee8>] (rtc_dev_ioctl+0x3a8/0x654)
[ 48.277529] [<c050dee8>] (rtc_dev_ioctl) from [<c019e310>] (do_vfs_ioctl+0xac/0x944)
[ 48.285291] [<c019e310>] (do_vfs_ioctl) from [<c019ebec>] (ksys_ioctl+0x44/0x68)
[ 48.292701] [<c019ebec>] (ksys_ioctl) from [<c019ec28>] (sys_ioctl+0x18/0x1c)
[ 48.299851] [<c019ec28>] (sys_ioctl) from [<c0009000>] (ret_fast_syscall+0x0/0x28)
```





## TRAVAMENTOS (cont.)

```
$ arm-linux-addr2line -f -e vmlinux 0xc0516eec  
snvs_rtc_set_time  
/opt/labs/ex/linux/drivers/rtc/rtc-snvs.c:140
```

```
$ arm-linux-gdb vmlinux  
(gdb) list *(snvs_rtc_set_time+0x60)  
0xc0516eec is in snvs_rtc_set_time (drivers/rtc/rtc-snvs.c:140).  
135  
136     dev_dbg(dev, "After conversion: %ld", time);  
137  
138     /* Disable RTC first */  
139     ret = snvs_rtc_enable(data, false);  
140     if (ret)  
141         return ret;  
142  
143     while(1);  
144
```





# DÚVIDAS?

E-mail      [sergio.prado@e-labworks.com](mailto:sergio.prado@e-labworks.com)  
Website    <http://e-labworks.com>



Embedded Labworks

