Debugging
Embedded Linux Kernel & Drivers

@ Kernel Space

Session 2: @ Kernel Space

Kernel Space	Purpose
printk & dmesg	Useful for monitoring
Module test cases	Useful for particular module test
/proc and /sys	Proc file system useful for overall system information. Sys file system useful for device hierarchy.
Probe Status	Useful for to test device working condition.
KDB	KDB is In built Debugger to test whenever kernel goes panic.
KGDB	KGDB is Kernel GDB useful for source level debugging.
Crash dump	Whenever kernel crashes, it create crash dump file for crash dump analysis.
ftrace	Ftrace is the ability to see what is happening inside the kernel

Kernel Probes

Kprobe & Jprobes

Kprobe

- Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively.
- You can trap at almost any kernel code address(*), specifying a handler.
- routine to be invoked when the breakpoint is hit.
- A kprobe can be inserted on virtually any instruction in the kernel.

Jprobe

- A jprobe is inserted at the entry to a kernel function, and provides convenient access to the function's arguments.
- A return probe fires when a specified function returns.

Interrupt Latency

How to measure interrupt latency

What is Interrupt Latency?

Interrupt latency is the time that elapses from when an interrupt is generated to when the source of the interrupt is serviced.

For many operating systems, devices are serviced as soon as the device's interrupt handler is executed.

Interrupt latency may be affected by microprocessor design, interrupt controllers, interrupt masking, and the operating system's (OS) interrupt handling methods

How to measure interrupt latency?

- To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument.
- The best tool to use is an oscilloscope.
- One approach is to use one pin on a GPIO interface to generate the interrupt. This pin can be monitored on the ‘scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled. The interval between the two signals may be easily read from the instrument.

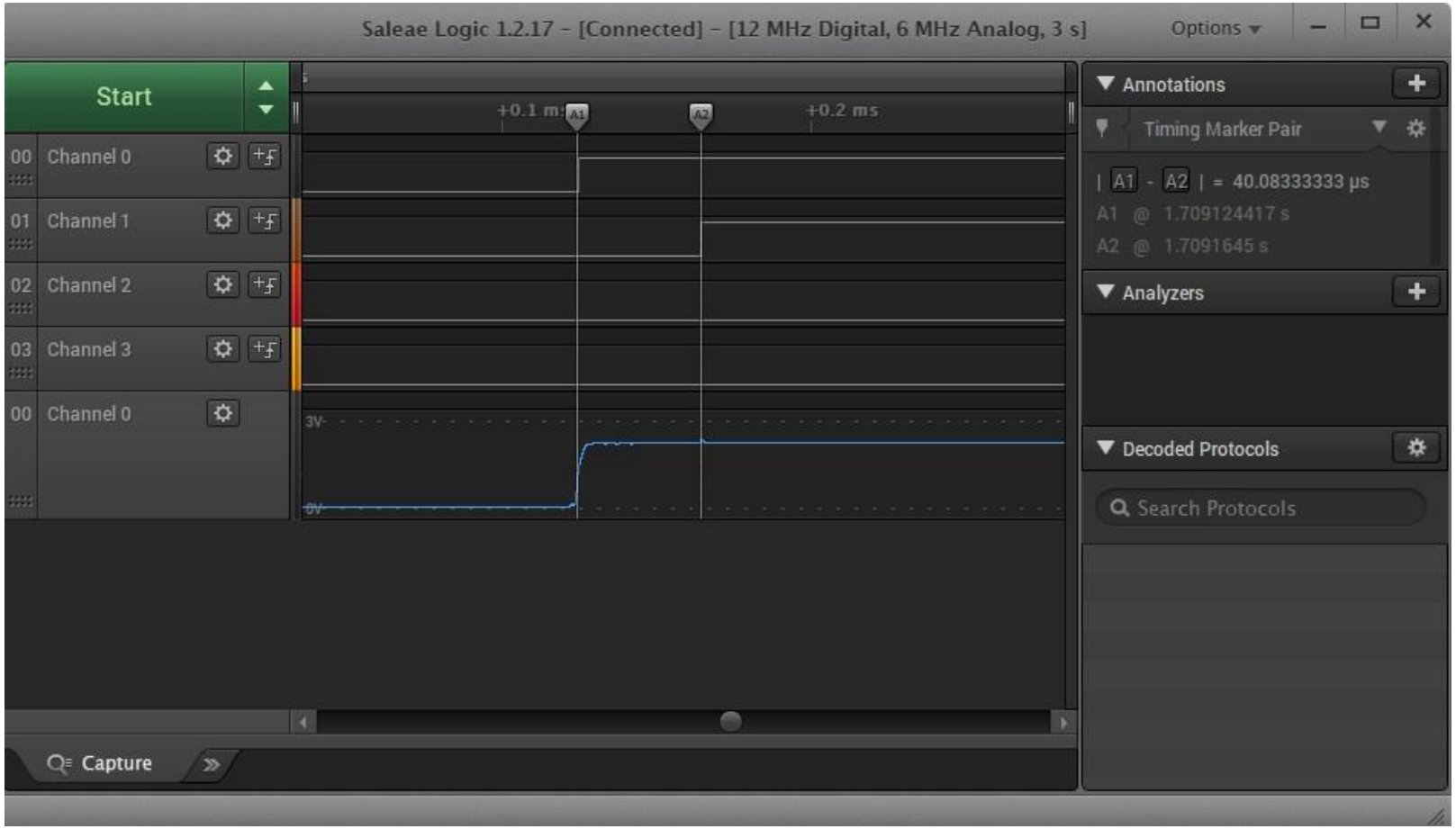
How to reduce interrupt latency?

- To reduce interrupt latency ARM Cortex M4 introduce new feature **“tail chain”**

  ARM Cortex M4 processor reduces interrupt latency using tail-chain.

- If system clock frequency is increases then instruction execution speed also increases. Once instruction execution speed increases to reduce interrupt latency.
  - To adjust system clock frequency using PLL.

Measure Interrupt Latency using Logic analyzer



Kernel Panic

Kernel Oops message Analysis

KERNEL OOPS

- **Kernel oops** is a way for the Linux kernel to communicate the user that a certain error has occurred.
- When the kernel detects a problem, it kills any offending processes and prints an oops message in the log, including the current system status and a **stack trace**.
- Different kind of errors could generate a kernel oops, including an illegal memory access or the execution of invalid instructions.
- The official Linux kernel documentation about handling oops messages is available at Documentation/admin-guide/bug-hunting.rst

KERNEL PANIC

- After a system has experienced an oops, some internal resources may no longer be operational.
- A kernel oops often leads to a kernel panic when the system attempts to use resources that have been lost.
- In a kernel panic, the execution of the kernel is interrupted and a message with the reason of the kernel panic is displayed in the kernel logs.

KERNEL OOPS Message

- Whenever kernel goes to panic stage it creates a OOPS message which contains following data:
  - BUG Summary: BUG: unable to handle kernel NULL pointer dereference at (null)
  - Which Function creates panic
  - Board Information
  - Which CPU runs incase of multicore processor
  - Kernel Version
  - Back Trace
  - CPU Registers & Flag registers
  - Processor Mode
  - Application PID
  - Application Name
  - Module Name

KERNEL OOPS -1

```
[   59.939827] pid:594 comm:insmod
[   59.943034] Unable to handle kernel NULL pointer dereference at virtual address 00000000
[   59.951393] pgd = fa05fd18
[   59.954121] [00000000] *pgd=00000000
[   59.957793] Internal error: Oops: 5 [#1] SMP ARM
Entering kdb (current=0xdd7d2e40, pid 594) on processor 0 Oops: (null)
due to oops @ 0xbf417034
CPU: 0 PID: 594 Comm: insmod Tainted: P         O      4.19.94-Kernel-Masters-g001db1705-dirty #2
Hardware name: Generic AM33XX (Flattened Device Tree)
PC is at panic_init+0x34/0x50 [panic]
LR is at panic_init+0x28/0x50 [panic]
pc : [<bf417034>]    lr : [<bf417028>]    psr: 600f0013
sp : dd5ebdc0  ip : 00000007  fp : c0b88948
r10: bf419000  r9 : 00000028  r8 : 00000000
r7 : bf417000  r6 : c0b88948  r5 : c0b8896c  r4 : c0c46100
r3 : 00000000  r2 : e90463b7  r1 : 00000007  r0 : bf418068
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
Control: 10c5387d  Table: 9b61c019  DAC: 00000051
CPU: 0 PID: 594 Comm: insmod Tainted: P         O      4.19.94-Kernel-Masters-g001db1705-dirty #2
```

KERNEL OOPS - 2

Hardware name: Generic AM33XX (Flattened Device Tree)
[<c001addc>] (unwind_backtrace) from [<c0014f2c>] (show_stack+0x10/0x14)
[<c0014f2c>] (show_stack) from [<c080a7cc>] (dump_stack+0xe4/0x11c)
[<c080a7cc>] (dump_stack) from [<c0131140>] (kdb_main_loop+0x328/0x7a0)
[<c0131140>] (kdb_main_loop) from [<c0133e2c>] (kdb_stub+0x214/0x444)
[<c0133e2c>] (kdb_stub) from [<c012a000>] (kgdb_cpu_enter+0x428/0x77c)
[<c012a000>] (kgdb_cpu_enter) from [<c012a534>] (kgdb_handle_exception+0xc4/0x220)
[<c012a534>] (kgdb_handle_exception) from [<c001a450>] (kgdb_notify+0x2c/0x64)
[<c001a450>] (kgdb_notify) from [<c006acec>] (notifier_call_chain+0x48/0x80)
[<c006acec>] (notifier_call_chain) from [<c006af64>] (__atomic_notifier_call_chain+0x70/0x120)
[<c006af64>] (__atomic_notifier_call_chain) from [<c006b0a0>] (notify_die+0x6c/0xc4)
[<c006b0a0>] (notify_die) from [<c0015094>] (die+0x164/0x368)
[<c0015094>] (die) from [<c001fa30>] (__do_kernel_fault.part.0+0x54/0x74)
[<c001fa30>] (__do_kernel_fault.part.0) from [<c0827b04>] (do_page_fault+0x3b8/0x3d0)
[<c0827b04>] (do_page_fault) from [<c001fb20>] (do_DataAbort+0x44/0xe4)
[<c001fb20>] (do_DataAbort) from [<c0009944>] (__dabt_svc+0x64/0xa0)

KERNEL OOPS - 3

```
Exception stack(0xdd5ebd70 to 0xdd5ebdb8)
bd60:                                     bf418068 00000007 e90463b7 00000000
bd80: c0c46100 c0b8896c c0b88948 bf417000 00000000 00000028 bf419000 c0b88948
bda0: 00000007 dd5ebdc0 bf417028 bf417034 600f0013 ffffffff
[<c0009944>] (__dabt_svc) from [<bf417034>] (panic_init+0x34/0x50 [panic])
[<bf417034>] (panic_init [panic]) from [<c000afd8>] (do_one_initcall+0x80/0x318)
[<c000afd8>] (do_one_initcall) from [<c00fb588>] (do_init_module+0x5c/0x1f8)
[<c00fb588>] (do_init_module) from [<c00fda10>] (load_module+0x2284/0x25a4)
[<c00fda10>] (load_module) from [<c00fdf98>] (sys_finit_module+0xbc/0xdc)
[<c00fdf98>] (sys_finit_module) from [<c0009000>] (ret_fast_syscall+0x0/0x28)
Exception stack(0xdd5ebfa8 to 0xdd5ebff0)
bfa0:                   4c4d7800 00000000 00000003 0044e7e0 00000000 beb60c68
bfc0: 4c4d7800 00000000 00000000 0000017b 00464560 00000000 beb60de8 00000000
bfe0: beb60c18 beb60c08 00446e41 b6d80d92
```

ADDR2LINE

- The addr2line tool is capable of converting a memory address into a line of source code:

Example 1:
arm-linux-gnueabihf-addr2line -f -e panic.ko panic_init
init_module
/home/km/debug/kernel/panic/panic.c:12

arm-linux-gnueabi-addr2line

$ arm-linux-gnueabihf-addr2line -f -e ~/debug/kernel/panic/panic.ko 0x34

init_module

/home/km/debug/kernel/panic/panic.c:14

Example 2:
$ arm-linux-addr2line -f -e vmlinux 0xc0539c44
mcp23sxx_spi_read
/home/sprado/elce/linux/drivers/pinctrl/pinctrl-mcp23s08.c:357

FADDR2LINE

- The faddr2line kernel script will translate a stack dump function offset into a source code line:

```
$ ./scripts/faddr2line vmlinux mcp23sxx_spi_read+0x34
mcp23sxx_spi_read+0x34/0x80:
mcp23sxx_spi_read at drivers/pinctrl/pinctrl-mcp23s08.c:357
```

panic_init+0x34/0x50

<fun_name>+<offset>/<fun_size>

GDB LIST

Example 1:

$ arm-linux-gdb vmlinux
(gdb) list *(mcp23sxx_spi_read+0x34)

```
0xc0539c44 is in mcp23sxx_spi_read (drivers/pinctrl/pinctrl-
mcp23s08.c:357).
352 u8 tx[2];
353
354 if (reg_size != 1)
355 return -EINVAL;
356
357 tx[0] = mcp->addr | 0x01;
358 tx[1] = *((u8 *) reg);
359
360 spi = to_spi_device(mcp->dev);
```

Example 2:

$ arm-linux-gnueabihf-gdb panic.ko

```
(gdb) list *(panic_init+0x34)
0x34 is in panic_init (/home/km/debug/kernel/panic/panic.c:14).
9
10      int *ptr=NULL;
11
12      static int panic_init(void) {
13              printk("pid:%d comm:%s\n",current->pid,current->comm);
14              printk("%d\n",*ptr); //creates kernel panic
15              return 0;
16      }
17
18      static void panic_exit(void) {
```

GDB DISASSEMBLE

Panic Example:

```
$ arm-linux-gnueabihf-gdb panic.ko

(gdb) disass panic_init
Dump of assembler code for function panic_init:
   0x00000000 <+0>:     andeq   r0, r0, r4
   0x00000004 <+4>:     andeq   r0, r0, r4, lsl r0
   0x00000008 <+8>:     andeq   r0, r0, r3
   0x0000000c <+12>:    subseq  r4, r5, r7, asr #28
   0x00000010 <+16>:    cmpne   r9, #-1073741816                ;
0xc0000008
   0x00000014 <+20>:    mcrcc   15, 2, pc, cr11, cr2, {7}       ;
<UNPREDICTABLE>
   0x00000018 <+24>:    rschi   r6, r4, #80, 16   ; 0x500000
   0x0000001c <+28>:    bhi     0xfe66abcc
   0x00000020 <+32>:    ldrhi   r8, [r6, #1237]!  ; 0x4d5
   0x00000024 <+36>:    bl      0x24 <panic_init+36>
   0x00000028 <+40>:    ldr     r3, [pc, #24]     ; 0x48 <panic_init+72>
   0x0000002c <+44>:    ldr     r0, [pc, #24]     ; 0x4c <panic_init+76>
   0x00000030 <+48>:    ldr     r3, [r3]
   0x00000034 <+52>:    ldr     r1, [r3]
   0x00000038 <+56>:    bl      0x38 <panic_init+56>
   0x0000003c <+60>:    mov     r0, #0
--Type <RET> for more, q to quit, c to continue without paging--
   0x00000040 <+64>:    pop     {r4, pc}
   0x00000044 <+68>:    andeq   r0, r0, r0
   0x00000048 <+72>:    andeq   r0, r0, r0
   0x0000004c <+76>:    andeq   r0, r0, r0, lsl r0
End of assembler dump.
```

```
(gdb) list *(0x00000034)
0x34 is in panic_init (/home/km/debug/kernel/panic/panic.c:14).
9
10      int *ptr=NULL;
11
12      static int panic_init(void) {
13              printk("pid:%d comm:%s\n",current->pid,current->comm);
14              printk("%d\n",*ptr); //creates kernel panic
15              return 0;
16      }
17
18      static void panic_exit(void) {
```

GDB DISASSEMBLE

$ arm-linux-gdb vmlinux

```
(gdb) disassemble /m mcp23sxx_spi_read
Dump of assembler code for function mcp23sxx_spi_read:
349 {
0xc0539c10 <+0>: mov r12, sp
0xc0539c14 <+4>: push {r4, r11, r12, lr, pc}
0xc0539c18 <+8>: sub r11, r12, #4
0xc0539c1c <+12>: sub sp, sp, #20
0xc0539c20 <+16>: push {lr} ; (str lr, [sp, #-4]!)
[...]
357 tx[0] = mcp->addr | 0x01;
0xc0539c3c <+44>: mov r0, #0
0xc0539c44 <+52>: ldrb r1, [r0]
0xc0539c54 <+68>: orr r1, r1, #1
0xc0539c58 <+72>: strb r1, [r11, #-26] ; 0xffffffe6
[...]
```

KDB/KGDB

Instruction level Debugger/Source level Debugger

What is KDB?

Kernel Debugger (kdb):

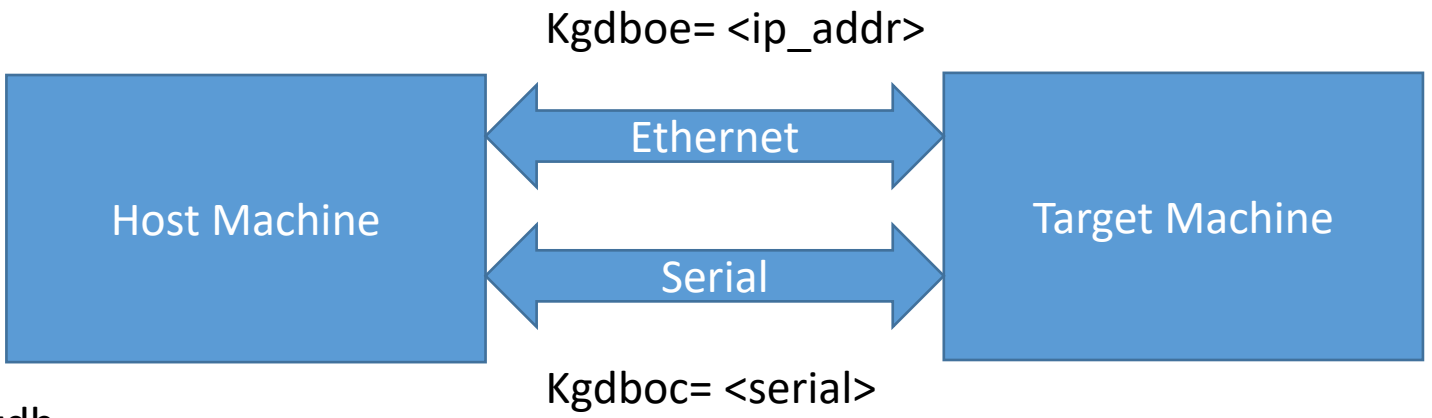
- Kdb is an instruction-level debugger used for debugging kernel code and device drivers.
- Before you can use it, you need to enable configuration options your kernel sources with kdb support and recompile the kernel.

What is KGDB?

- Kgdb is a source-level debugger.
- It is easier to use than kdb because you don't have to spend time correlating assembly code with your sources.
- However it's more difficult to set up because an additional machine is needed to front-end the debugging.
- gdb runs on the host machine, while the kgdb-patched kernel runs on the target hardware.
- The host and the target are connected via a serial null-modem cable .

What is KGDB?

- Two machines are required for using kgdb.
- One of these machines is a development machine and the other is the target machine. The kernel to be debugged runs on the target machine.
- The development machine runs an instance of gdb against the vmlinux file which contains the symbols (not boot image such as bzImage, zImage, uImage...).
- In gdb the developer specifies the connection parameters and connects to kgdb



Host running gdb
$ arm-linux-gnueabihf-gdb vmlinux
(gdb) b printk
(gdb) c

Target running a kernel patched with kgdb

KDB vs KGDB

Parameter	KDB	KGDB
Advantage	KDB easy to set up, because you don't need an additional machine to do the debugging	KGDB more difficult to set up because an additional machine is needed to front-end the debugging.
Disadvantage	you need to correlate your sources with disassembled code	Don't have to spend time correlating assembly code with your sources
Debugger Environment	It is a debugger that needs to be built inside the kernel. All it requires is a console using which commands can be entered and output displayed on the console.	It requires a development machine to run the debugger as a normal process that communicates with the target using the GDB protocol over a serial cable. Recent versions of KGDB support the Ethernet interface.

KDB vs KGDB

Parameter	KDB	KGDB
Support for source-level debugging	No support for source-level debugging	Support for source-level debugging provided the kernel is compiled with the -g flag on the development machine and the kernel source tree is available. On the development machine where the debugger application runs, -g option tells gcc to generate debugging information while compiling, which in conjunction with source files provides source-level debugging.

KDB vs KGDB

Parameter	KDB	KGDB
Kernel module debugging	KDB provides support for kernel module debugging.	Debugging modules using KGDB is tricky because the module is loaded on the target machine and the debugger (GDB) runs on a different machine; so the KGDB debugger needs to be informed of the module load address. KGDB 1.9 is accompanied by a special GDB that can automatically detect module loading and unloading. For KGDB versions equal to or less than 1.8, the developer has to make use of an explicit GDB command add symbol-file to load the module object into GDB's memory along with the module load address.

KDB vs KGDB

Parameter	KDB	KGDB
Debugging features offered	The most commonly used debugging features of KDB are: displaying and modifying memory and registers applying breakpoints stack backtrace Along with the user-applied breakpoints, KDB is invoked when the kernel hits an irrecoverable error condition such as panic or OOPS. The user can use the output of KDB to diagnose the problem	Supports GDB execution control commands, stack trace, and KGDB-specific watchpoints among a host of other feature such as thread analysis.

LKCD Analysis

Linux Kernel Crash Dump [kexec, kdump]

Oh, Customer got a problem

- Dump image is useful for post-crash analysis
  - A snapshot on critical kernel error (panic)
  - You can see the kernel state via crash, gdb, ...
- Different dump methods: kdump, LKCD, ...

kdump

- Kdump uses kexec to quickly boot to a dump-capture kernel whenever a dump of the system kernel's memory needs to be taken (for example, when the system panics).
- When the system kernel boots, we need to reserve a small section of memory for the dump-capture kernel, passing a parameter via kernel command line. crashkernel=64M.
- Using the kexec -p command from kexec-tools we can load the dump-capture kernel into this reserved memory.

kdump

- On a kernel panic, the new kernel will boot and you can access the memory image of the crashed kernel through /proc/vmcore.
- This exports the dump as an ELF-format file that can be copied and analysed with tools such as GDB and crash.
- More information is available in the Linux kernel source code at Documentation/kdump/kdump.txt.

kdump

- Integrated in mainline kernel
- Standard on SLES10 i386, x86-64 and ppc64
- Reboot-based dump mechanism
  - More robustness and flexibility
- Requires more resources
  - A dedicated dump kernel binary
  - A fixed memory area for 2nd kernel
- Cannot dump non-disruptively

kdump

- A secondary (crash) kernel is started after crash
- Kexec is used for kernel-to-kernel switch
- The crash kernel runs in a reserved area
  - The old kernel memory is preserved & untouched
  - ELF image accessible via /proc/vmcore
  - Raw image accessible via /dev/oldmem
- Dump is done on the capture kernel context
  - Devices are re-initialized to sane state
  - You can do almost everything there..

kexec

- Kexec is a fastboot mechanism that allows booting a Linux kernel from the context of an already running kernel without going through BIOS.
- BIOS can be very time consuming, especially on big servers with numerous peripherals.
- This can save a lot of time for developers who end up booting a machine numerous times.

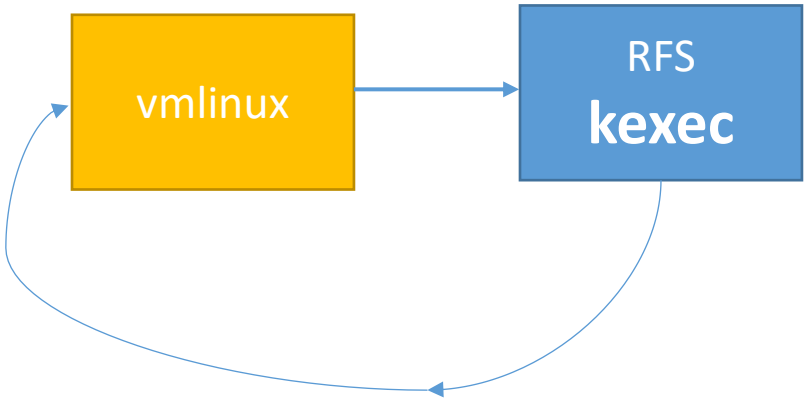
Normal boot vs kexec boot

1st Boot Process: power on reset

Basic h/w initialization done here

Kexec Boot Process: 2nd process

ROM, X-loader, u-boot

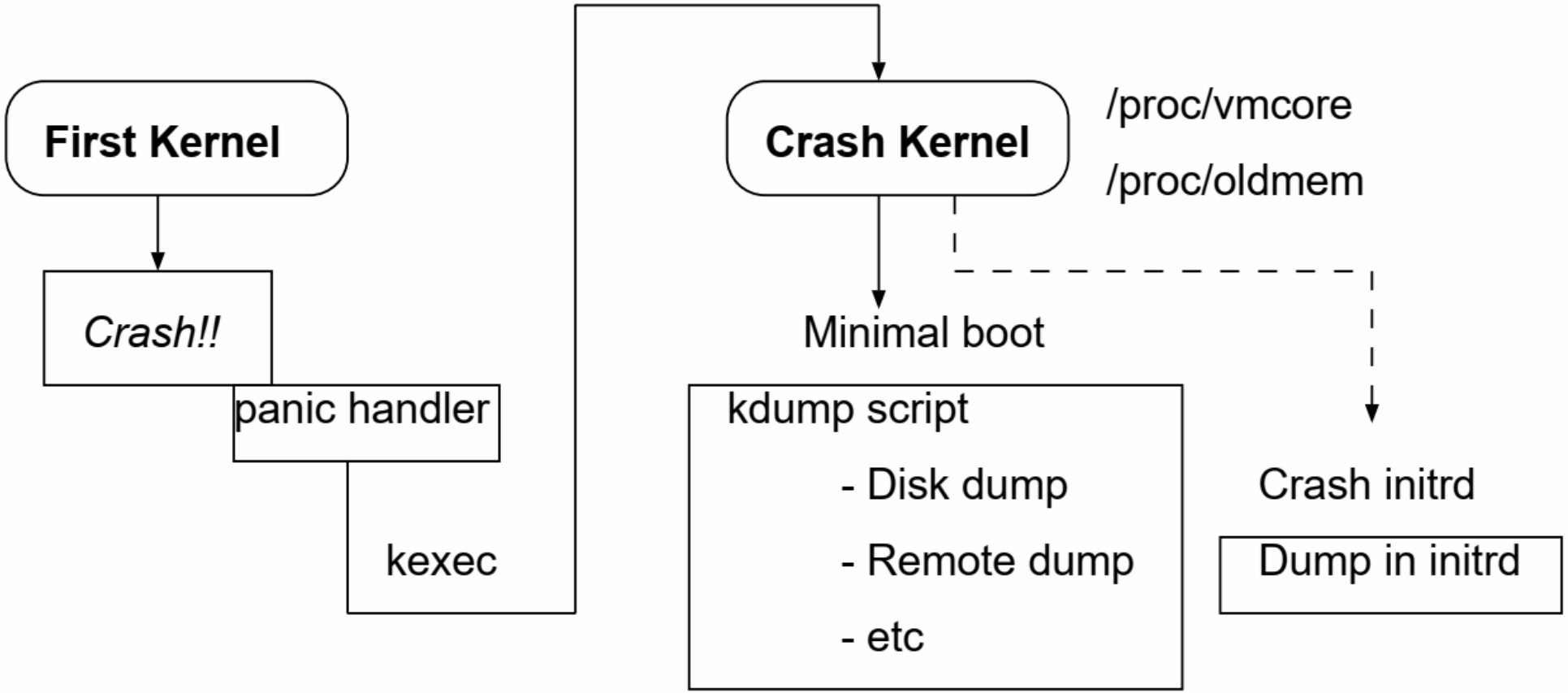


Kexec examples

- **sudo kexec -l /boot/vmlinuz-4.19.132 --append="BOOT_IMAGE=/boot/vmlinuz-4.19.132 root=UUID=f51c4fff-68ae-4bb9-86dd-5d8716be2821 ro quiet splash" --initrd=/boot/initrd.img-4.19.132**

- **sudo kexec -l /boot/vmlinuz-4.19.94-Kernel-Masters-g001db1705-dirty --append="console=ttyO0,115200n8 root=/dev/mmcblk1p1 rootfstype=ext4 rootwait"**
-

LKCD Design



LKCD Design

- **Crashkernel = [ramsize begin - end]<sizeofimage>@[location]**
- **Example:**

- **Crashkernel=32M**