To summarize, the Linux Armv8 exception vector table routing process is shown in the flowchart below.

As said in the Armv8 manual, `vbar_el1` stores the base address of exception vector table, which is initialized inside the `__primary_switched` function right before calling `start_kernel`.

From the code above we can infer that the vector table is called `vectors`, which is exactly defined in `arch/arm64/kernel/entry.S`. 
The `kernel_ventry` macro is also defined inside `entry.S`, which also does various security checks and detects stack overflow (if `CONF_VMAP_STACK` is enabled, part of a kernel feature known as Virtually Mapped Kernel Stacks).

- Inside this macro, the real handler `el\el\ht\()\regsize\()\label\()_handler` is invoked
- As seen above, the handler will first call `kernel_entry`, which is defined as follows

```assembly
.macro kernel_entry, el, regsize = 64
  .if \regsize == 32
    mov w0, w0       // zero upper 32 bits of x0
  .endif
  stp x0, x1, [sp, #16 * 0]
  stp x2, x3, [sp, #16 * 1]
  stp x4, x5, [sp, #16 * 2]
  stp x6, x7, [sp, #16 * 3]
  stp x8, x9, [sp, #16 * 4]
...
```

which will store the general-purpose registers and clear them. There is also a `kernel_exit` which will restore general-purpose register, etc.

- The actual exception handler is in `arch/arm64/kernel/entry-common.c`

```assembly
Exception return is achieved through calling `kernel_exit`, which will execute `ERET` to restore PC and PSTATE
```
The General Process of Exception Handling

- Below is a flow chart of what will happen during exception handling (using system call as an example)

```assembly
.in entry.s
    .macro entry_handler_el:req, ht:req, regsize:req, label:req
    SYM_CODE_START_LOCAL(ret_to_kernel)
    kernel_entry el, regsize
    mov x0, sp
    bl el\(\ht\)\_regsize\(\ht\)\_label\(\ht\)\_handler
    .if \el == 0
    b ret_to_user
    .else
    b ret_to_kernel
    .endm
    SYM_CODE_END(ret_to_kernel)

    .macro kernel_exit_el
    if \el == 0
    disable_dai
    .endm

    if \el == 0
    exit_to_kernel_mode
    .endm
```

Exception Handlings Specifics

- All the actual exception handling happens in the file `arch/arm64/kernel/entry-common.c`
- There are several important helper functions and macros to be identified
  - `entry_from_kernel_mode, exit_to_kernel_mode, arm64_enter_nmi, arm64_exit_nmi`: some setup functions

- If aarch32 is selected and enabled, the process is similar except that the top-level as well as specific handler will be the 32-bit version equivalent
- do_interrupt_handler: execute interrupt handler using the function pointer being passed in.
- panic_unhandled: panic the kernel and print "Unhandled exception"
- \#define UNHANDLED\(el, regsize, vector\) \ asmlinkage void noinstr el##_##regsize##_##vector##_handler(struct pt_regs *regs) \ { \ const char *desc = #regsize "-bit " #el " " #vector; \ __panic_unhandled(regs, desc, read_sysreg(esr_el1)); \ }

- el1Interrupt: called when dealing with irq and fiq exceptions taken from current level like below. The handler_arch_irq and handler_a
  arch_fiq are two function pointers that will be initialized by the irq driver. (See the drivers/irqchip/) The el1_interrupt function
  will first mask the irq and fiq interrupts, then execute the interrupt handler. (either handle_arch_irq or handle_arch_fiq)
  \ asmlinkage void noinstr el1h_64_irq_handler(struct pt_regs *regs) \ { \ el1_interrupt(regs, handle_arch_irq); \ }
  \ asmlinkage void noinstr el1h_64_fiq_handler(struct pt_regs *regs) \ { \ el1_interrupt(regs, handle_arch_fiq); \ }
  \ static void noinstr el1_interrupt(struct pt_regs *regs, void (*handler)(struct pt_regs *)) \ { write_sysreg(DAIF_PROCCTX_NOIRQ, daif); enter_el1_irq_or_nmi(regs); do_interrupt_handler(regs, handler); /* Note: thread_info::preempt_count includes both thread_info::count
   * and thread_info::need_resched, and is not equivalent to
   * preempt_count() . */
   if (IS_ENABLED(CONFIG_PREEMPTION) &&
    READ_ONCE(current_thread_info()->preempt_count) == 0)
    arm64_preempt_schedule_irq();
   exit_el1_irq_or_nmi(regs); }

- enter_from_user_mode, exit_to_user_mode: setup functions just like enter_from_kernel_mode/exit_to_kernel_mode
- el0Interrupt: similar to el0Interrupt but do not contain codes related to non-maskable interrupt
- handle_bad_stack: panic the kernel and print out FAR and ESR info. Called by \_bad_stack in entry.S

- el1Interrupt: similar to el0Interrupt but do not contain codes related to non-maskable interrupt

- all synchronous exceptions are handled similarly: first extract the content of ESR_EL1 register, then check the Exception Class field to
determine the nature of the exception, and then call the corresponding sub-handlers. All the sub-handlers have similar patterns as well:
  - enter_from_user_mode: do the actual handling
  - local_daif_inherit: will handler state tracking on ordinary interrupt entries (more explanation in the comment in
include/linux/entry-common.h)
  - local_daif_mask: will restore the DAIF bits that were modified by taking an exception
  - exit_to_kernel_mode: will handle return from exception

- All irq and fiq exceptions are handled using the el0/1_interrupt do_interrupt_handler

- All SErrors are handled similarly
First read the ESR register, then return DAIF flag
Then execute do_serror

System Call Handling

- System call is invoked via SVC #<imm> instruction. It is a synchronous exception handled by e10t_64_sync_handler and its Exception Class is ESR_ELx_EC_SVC64, which then invokes e10_svc function in arch/arm64/kernel/syscall.c. The e10_svc then invoke e10_svc_common and invoke_syscall will execute the actual system call given the system call number and the system call table. Full process is shown below
  - program makes a system call C libraries such as glibc then gets invoked SVC #<imm> e10t_64_sync_handler e10_svc e10_svc_common invoke_syscall __invoke_syscall (which is just a wrapper for return syscallfn(regs))
  - aarch32 will handled similarly but with equivalent compat functions
- Investigation about glibc/musl is documented in this confluence page