Hunting the bugs at compile time

- It is always good to catch some odd changes to the code at compile time when it is known that only subset of options is valid and adding something other than is supported without changing implementation is known to be buggy.
- There is a number of build time checks performed by preprocessor, compiler and semantics parsers like sparse, however neither of them can catch case described above.
- For that purpose Linux has number of preprocessor macros, that evaluate (if possible, otherwise it is illegal to use them) condition at build time and raise build error.

Hunting the bugs at compile time (cont.)

- These routines are defined in `<linux/bug.h>`
  - **BUILD_BUG**() - stop build unconditionally.
  - **BUILD_BUG_ON**(condition) - stop build when @condition evaluates to *true*. Note that @condition must be compile time evaluable (e.g. integer constant, known pointer value, and any expression with values known at compile time).
  - **BUILD_BUG_ON_ZERO**(condition) - stop build when @condition is *true*. Can be used as structure member initializer and returns (size_t) 0 if @condition is *false*.
  - **BUILD_BUG_ON_NULL**(condition) - stop build when @condition is *true*, Can be used as structure member initializer and returns NULL if @condition is *false*.
  - **BUILD_BUG_ON_NOT_POWER_OF_2**(expr) - stop build if @expr isn’t power of two.

Hunting the bugs at compile time (cont.)

- Let’s look at their functionality by examples

```
static inline void dst_hold(struct dst_entry *dst)
{
    /*
    * If your kernel compilation stops here, please check
    * __pad_to_align_refcnt declaration in struct dst_entry
    */
    BUILD_BUG_ON(offsetof(struct dst_entry, __refcnt) & 63);
    atomic_inc(&dst->__refcnt);
}
```

- Here is compilation would stop if offset of __refcnt field in struct dst_entry isn’t aligned to 64 bytes (i.e. bits 0-5, 2^6 - 1 == 63 should be zero).

Hunting the bugs at compile time (cont.)

- There are other hints on bug hunting at compile time. With some of them you should be already familiar from previous lecture.
  - Do not ignore compiler/preprocessor warnings: they pointing you to potential problems within your code
  - Use extra warning levels by passing W=1..3 option to make
  - Use sparse(1). This gives you even more warnings you can’t expect from compiler/preprocessor
  - Use static analysis and report generation options like `make check_stack`
  - Take look at → Kernel hacking → Compile-time checks and compiler options in during kernel configuration
  - Build source using different gcc versions or even use different compiler (e.g. clang, icc) they might give you more points for debug.

OOPS and Panic

- **OOPS** it is runtime, **non-fatal** condition happening in response to some unspecified/unhandled behaviour in functionality or triggered by external events during the system operation (e.g. hotplug, process kill due to OOM). System can either recover from such condition without any data/functionality loss and continue to run, even with limited functionality.
- **Panic** is a **fatal** condition, where system can not continue operations without significant functionality and/or data loss nor it can recover from such conditional safely. Often after system recovers from certain non-fatal OOPS it might trigger panic later as normal system functionality is compromised, some vital data structures might be corrupted.

OOPS and Panic (cont.)

What Linux actually does on OOPS?

- It is architecture dependent
- In general there are traps configured for exceptions like page fault, general protection, invalid opcode, divide by zero, alignment error (for platforms with strict alignment rules like some SPARC), etc.
- Then, depending on trap handlers implementation it calls `notify_die()` to call registered with `register_die_notifier()` notifiers (callback functions) that inform it's subscribers about exception (one of such subscribers is kgdb)
- Finally, if exception related to user mode, it is translated to signal (e.g. SIGSEGV). Overwise it triggers architecture dependent die() call which makes informative reports we see on OOPS.

OOPS and Panic (cont.)

Why and how to control behaviour on OOPS?
OOPS may cause system to panic immediately, rather than continue

- To catch problem at first place (e.g. paging fault at NULL pointer dereference)
- Eases debugging process
- By using oops=panic on Linux cmdline from bootloader one can instruct kernel to treat OOPS as fatal error and thus panic.
- It is also possible to control value of this parameter at runtime via sysctl interface in proc filesystem.

```
~# sysctl -w kernel.panic_on_oops = 1
```

On the same side as OOPS/Panic: warnings, traces

- There is another class conditions, which are similar to OOPS in sense they are not fatal nor even may indicate any recovery made/needed by/from the system nor user to address them.
- They typically might indicate some misconfiguration, misunderstanding in use of some kernel APIs/interfaces, etc.
- They have different from OOPS/Panic nature and usually added to code with means like WARN(), WARN_ON(), WARN_ONCE(), WARN_ON_ONCE() and dump_stack() in places where attention is needed due to potential problems using approach.

On the same side as OOPS/Panic: warnings, traces

Runtime warning routines

- They are defined as generic in `<asm-generic/bug.h>` as preprocessor macros and might be overwritten by arch specific code (WARN_ON() currently).
  - **WARN**(condition, format...) - trigger warning message when conditional evaluates to true. Use printk() for message to print @format with optional arguments message
  - **WARN_ON**(condition) - trigger warning message when conditional evaluates to true
  - **WARN_ONCE**(condition, format...) - like WARN(), but print warning message only once
  - **WARN_ON_ONCE**(condition) - like WARN_ON(), but print warning message only once
- They might be used in conditional statements because they return either 0 when warning isn't triggered or 1 when it is. They might just check for conditional if CONFIG_BUG isn't enabled.

```
if (WARN_ON_ONCE((rt->dst.flags & DST_NOCACHE) &&
                 !atomic_read(&rt->dst.__refcnt)))
    return -EINVAL;
```

Stack trace dump routines

- It might be required to dump stack trace at the same error code path. For example, when implementing custom BUG()/WARN() functionality that does not depend on CONFIG_BUG or have to add custom title before trace message. It might depend on some other conditions like CONFIG_FOO for current driver/module or always present to let user runtime checks.
- One of the good examples of such use of dump_stack() is

```
#define ASSERT_RTNL() do { \
    if (unlikely(!rtnl_is_locked())) { \
      printk(KERN_ERR "RTNL: assertion failed at %s (%d)\n", \
          __FILE__,  __LINE__); \
      dump_stack(); \
    } \
} while(0)
```

Trigger OOPS/Panic from kernel code

- here are two preprocessor defines in `<asm-generic/bug.h>`, that use `panic()` in their implementation:
  - BUG() trigger panic unconditionally
  - BUG_ON(condition) trigger panic when @condition evaluates to *true*

```
#define BUG() do { \
    printk("BUG: failure at %s:%d/%s()!\n", __FILE__, __LINE__, __func__); \
    panic("BUG!"); \
} while (0)

#define BUG_ON(condition) do { if (unlikely(condition)) BUG(); } while (0)
```

- Note that arch code might provide overrides for both BUG() and BUG_ON() and thus it is not necessary that panic() is called on BUG(): it might be just OOPS.
- Also from user space

```
~# echo ‘c’ >/proc/sysrq-trigger
```

OOPS message format

- In general it is architecture dependent, but have some generic structure
- It might contain following data for tracing to problem origin
  - Register contents
  - Stack back traces
  - Strings describing hardware where message is triggered
  - String describing type of the problem (e.g. page fault, division by zero, etc)
  - List of modules linked to the kernel
  - Position of Instruction Pointer (IP) and symbolic name of function owning it
  - Various values from Linux Kernel specific data structures (e.g. page directory entry (pde), page table entry (pte), etc).

OOPS message format (cont.)

Let’s generate it

```
struct time_data {
    ktime_t start_time;          /* 0 */
    struct list_head list;       /* 8 */
};

__attribute__((__noinline__))
static void add_to_list(struct list_head *node)
{
    BUG_ON(!node);               /* It does not trigger oops because node is not NULL! */
    list_add_tail(node, &time_list);
}

static int __init hello_init(void)
{
    struct time_data *time_data = 0; // kmalloc(sizeof(*time_data), GFP_KERNEL);

    add_to_list(&time_data->list);
    /* … */
    return 0;
};
```

OOPS message format (cont.)

Changes in Makefile

```
ifneq ($(KERNELRELEASE),)
# kbuild part of makefile
obj-m  := hello.o
ccflags-y += -g                                 # add debugging info
else
# normal makefile
KDIR ?= /lib/modules/`uname -r`/build

default:
    $(MAKE) -C $(KDIR) M=$$PWD
    cp hello.ko hello.ko.unstripped
    $(CROSS_COMPILE)strip -g hello.ko    # strip only debugging info

clean:
    $(MAKE) -C $(KDIR) M=$$PWD clean

%.s %.i: %.c                                    # just use make hello.s instead of objdump
    $(MAKE) -C $(KDIR) M=$$PWD $@

endif
```

OOPS message format (cont.)

```
~ # insmod hello.ko
[  13.704625] hello: loading out-of-tree module taints kernel.
[  13.711132] Unable to handle kernel NULL pointer dereference at virtual address 00000008
[  13.719693] pgd = d6388774
[  13.722518] [00000008] *pgd=9a2d9831, *pte=00000000, *ppte=00000000
[  13.729100] Internal error: Oops: 817 [#1] SMP ARM
[  13.734105] Modules linked in: hello(O+)
[  13.738208] CPU: 0 PID: 78 Comm: insmod Tainted: G        O    4.19.114 #5
[  13.745834] Hardware name: Generic AM33XX (Flattened Device Tree)
[  13.752206] PC is at add_to_list+0x1c/0x2c [hello]
[  13.757207] LR is at hello_init+0xc/0x1000 [hello]
[  13.762205] pc : [<bf00001c>]    lr : [<bf00500c>]    psr: 200f0013
[  13.768743] sp : da255dc8  ip : da237ac0  fp : 00000000
[  13.774192] r10: bf002040  r9 : c1704c48  r8 : 00000000
[  13.779644] r7 : bf005000  r6 : ffffe000  r5 : c1704c48  r4 : c1888140
[  13.786455] r3 : bf002000  r2 : bf002000  r1 : 00005782  r0 : 00000008
[  13.793267] Flags: nzCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment none
[  13.800711] Control: 10c5387d  Table: 9a258019  DAC: 00000051
[  13.806707] Process insmod (pid: 78, stack limit = 0x7fb7c0e3)
[  14.022583] Exception stack(0xda255fa8 to 0xda255ff0)
[  13.817342] 5dc0:                   c1888140 c0302d70 db16a400 00000000 00210d00 da255ddc
[  13.825880] 5de0: c1704c48 da2dc840 8040003f bf002088 bf002088 51b4471b ffe00000 da2dc8c0
```

OOPS message format (cont.)

```
[  13.812792] Stack: (0xda255dc8 to 0xda256000)
[  13.817342] 5dc0:                   c1888140 c0302d70 db16a400 00000000 00210d00 da255ddc
[  13.825880] 5de0: c1704c48 da2dc840 8040003f bf002088 bf002088 51b4471b ffe00000 da2dc8c0
[  13.834417] 5e00: 8040003e e1261000 da2dc8c0 51b4471b e1261000 da2dc840 bf002040 51b4471b
...
[  13.953953] 5fc0: b6f0738c b6f26950 000013a4 00000080 00000001 bec54e8c 001086c4 000f411e
[  13.962491] 5fe0: bec54b48 bec54b38 0003b270 b6de01b0 600f0010 0011b1d8 00000000 00000000
[  13.971046] [<bf00001c>] (add_to_list [hello]) from [<bf00500c>] (hello_init+0xc/0x1000
[hello])
[  13.980233] [<bf00500c>] (hello_init [hello]) from[<c0302d70>] (do_one_initcall+0x54/0x208)
[  13.989060] [<c0302d70>] (do_one_initcall) from [<c03d6160>] (do_init_module+0x64/0x214)
[  13.997513] [<c03d6160>] (do_init_module) from [<c03d8654>] (load_module+0x22dc/0x2628)
[  14.005871] [<c03d8654>] (load_module) from [<c03d8b10>] (sys_init_module+0x170/0x1c4)
[  14.014139] [<c03d8b10>] (sys_init_module) from [<c0301000>] (ret_fast_syscall+0x0/0x54)
[  14.022583] Exception stack(0xda255fa8 to 0xda255ff0)
[  14.027856] 5fa0:                   b6f0738c b6f26950 0011b1d8 000013a4 001086c4 00000000
[  14.036395] 5fc0: b6f0738c b6f26950 000013a4 00000080 00000001 bec54e8c 001086c4 000f411e
[  14.044930] 5fe0: bec54b48 bec54b38 0003b270 b6de01b0
[  14.050203] Code: e3023000 e34b3f00 e5932004 e5830004 (e5803000)
[  14.056636] ---[ end trace 9ddb61d277c49039 ]---
Segmentation fault
```

OOPS message format (cont.)

Let's find it (objdump): `PC is at add_to_list+0x1c/0x2c`

```
$ ${CROSS_COMPILE}objdump -dS hello.ko.unstripped | less
00000000 <add_to_list>:
__attribute__((__noinline__)) static void add_to_list(struct list_head *node)
{
      BUG_ON(!node);
   0:   e3500000        cmp   r0, #0
   4:   1a000000        bne   c <add_to_list+0xc>
   8:   e7f001f2        .word   0xe7f001f2
...
static inline void list_add_tail(struct list_head *new, struct list_head *head)
{
      __list_add(new, head->prev, head);
   c:   e3003000        movw r3, #0
  10:   e3403000        movt r3, #0
  14:   e5932004        ldr   r2, [r3, #4]
      next->prev = new;
  18:   e5830004        str   r0, [r3, #4]
      new->next = next;
  1c:   e5803000        str   r3, [r0]
      new->prev = prev;
  20:   e5802004        str   r2, [r0, #4]
```

OOPS message format (cont.)

Let's find it (gdb): `PC is at add_to_list+0x1c/0x2c`

```
$ ${CROSS_COMPILE}gdb -q hello.o
(gdb) disas add_to_list+0x0c,add_to_list+0x1c
Dump of assembler code from 0x18 to 0x28:
   0x00000018 <add_to_list+12>:    movw    r3, #0
   0x0000001c <add_to_list+16>:    movt    r3, #0
   0x00000020 <add_to_list+20>:    ldr    r2, [r3, #4]
   0x00000024 <add_to_list+24>:    str    r0, [r3, #4]
End of assembler dump.
(gdb) quit
$
```

OOPS message format (cont.)

THUMB2 mode (CONFIG_THUMB2_KERNEL)

```
~ # insmod hello.ko
[  35.117134] hello: loading out-of-tree module taints kernel.
[  35.123766] Unable to handle kernel NULL pointer dereference at virtual address 00000008
[  35.132332] pgd = 83ba745f
[  35.135165] [00000008] *pgd=9a294831, *pte=00000000, *ppte=00000000
[  35.141743] Internal error: Oops: 817 [#1] SMP THUMB2
[  35.147034] Modules linked in: hello(O+)
[  35.151152] CPU: 0 PID: 81 Comm: insmod Tainted: G        O    4.19.114 #6
[  35.158807] Hardware name: Generic AM33XX (Flattened Device Tree)
[  35.165201] PC is at hello_init+0x21/0xfff [hello]
[  35.170228] LR is at do_one_initcall+0x3f/0x16c
[  35.174971] pc : [<bf805022>]    lr : [<c0302ac3>]   psr: 200f0033
[  35.181533] sp : da2d3db8  ip : da27b240  fp : c1404c48
[  35.187003] r10: da27bdc8  r9 : bf802040  r8 : 00000000
[  35.192474] r7 : bf805001  r6 : ffffe000  r5 : 00000008  r4 : 00000000
[  35.199309] r3 : bf802000  r2 : bf802240  r1 : bf802000  r0 : bf801050
...
...
[   35.463474] Code: 0050 605d f6cb 7080 (60a3) 60e1
```