Don't blame devres - devm_kzalloc() is not harmful

Use-after-free bugs in drivers and what to do about them.

Bartosz Golaszewski
Linaro
FOSDEM 2023

Without devres

```
struct xyz_data {
        struct foo *foo;
        struct bar *bar;
        struct baz *baz;
};
```

```
int xyz_remove(struct platform_device *pdev)
{
        struct xyz_data *xyz = platform_get_drvdata(pdev);

        free_baz(xyz->baz);
        free_bar(xyz->bar);
        free_foo(xyz->foo);
        kfree(xyz);

        return 0;
}
```

```
int xyz_probe(struct platform_device *pdev)
{
        struct xyz_data *xyz;

        xyz = kzalloc(sizeof(*zyx), GFP_KERNEL);
        if (!xyz)
                return -ENOMEM;

        xyz->foo = alloc_foo();
        if (IS_ERR(xyz->foo)) {
                kfree(xyz);
                return PTR_ERR(xyz->foo);
        }

        xyz->bar = alloc_bar();
        if (IS_ERR(xyz->bar)) {
                free_foo(xyz->foo);
                kfree(xyz);
                return PTR_ERR(xyz->bar);
        }

        xyz->baz = alloc_baz();
        if (IS_ERR(xyz->baz)) {
                free_bar(xyz->bar);
                free_foo(xyz->foo);
                kfree(xyz);
                return PTR_ERR(xyz->baz);
        }

        platform_set_drvdata(pdev, xyz);

        return 0;
}
```

Without devres (alternatively)

```
struct xyz_data {
        struct foo *foo;
        struct bar *bar;
        struct baz *baz;
};
```

```
int xyz_remove(struct platform_device *pdev)
{
        struct xyz_data *xyz = platform_get_drvdata(pdev);

        free_baz(xyz->baz);
        free_bar(xyz->bar);
        free_foo(xyz->foo);
        kfree(xyz);

        return 0;
}
```

```
int xyz_probe(struct platform_device *pdev)
{
        struct xyz_data *xyz;
        int ret = 0;

        xyz = kzalloc(sizeof(*zyx), GFP_KERNEL);
        if (!xyz)
                return -ENOMEM;

        xyz->foo = alloc_foo();
        if (IS_ERR(xyz->foo)) {
                ret = PTR_ERR(xyz->foo);
                goto err_free_xyz;
        }

        xyz->bar = alloc_bar();
        if (IS_ERR(xyz->bar)) {
                ret = PTR_ERR(bar);
                goto err_free_foo;
        }

        xyz->baz = alloc_baz();
        if (IS_ERR(xyz->baz)) {
                ret = PTR_ERR(baz);
                goto err_free_bar;
        }

        platform_set_drvdata(pdev, xyz);

        return 0;

err_free_bar:
        free_bar(xyz->bar);
err_free_foo:
        free_foo(xyz->foo);
err_free_xyz:
        kfree(xyz);
        return ret;
}
```

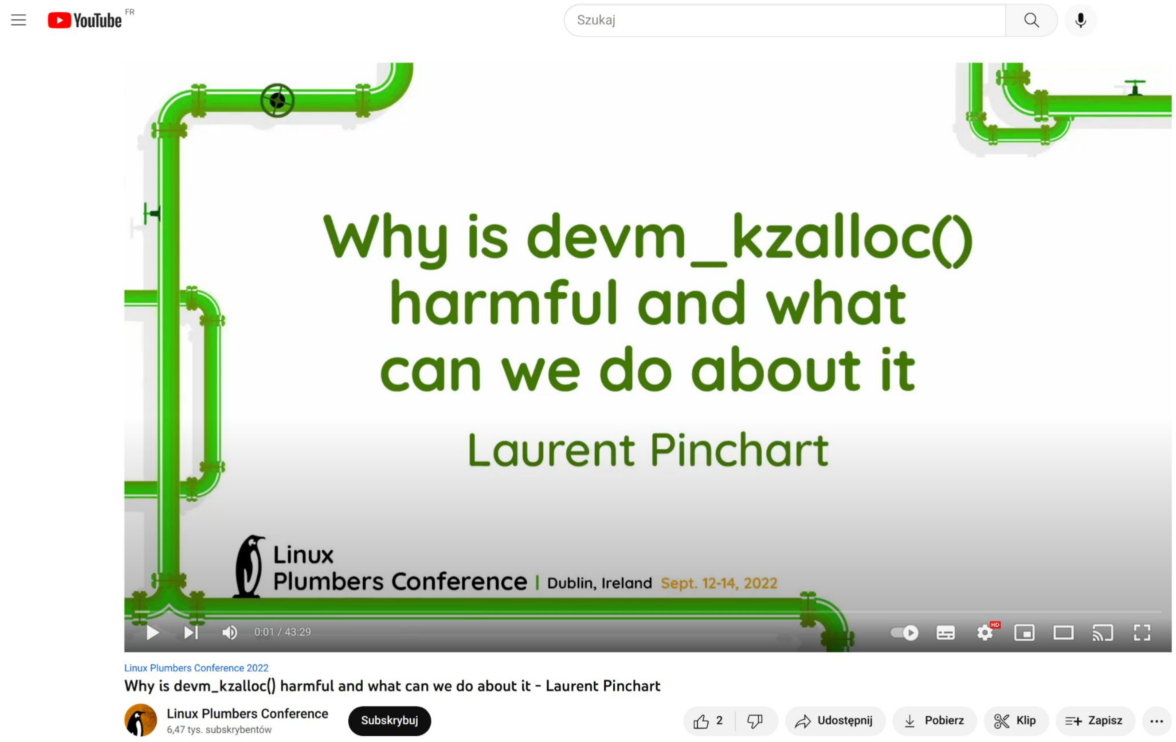
With devres

```
struct xyz_data {
        struct foo *foo;
        struct bar *bar;
        struct baz *baz;
};
```

```
int xyz_remove(struct platform_device *pdev)
{
        struct xyz_data *xyz = platform_get_drvdata(pdev);

        free_baz(xyz->baz);
        free_bar(xyz->bar);
        free_foo(xyz->foo);
        kfree(xyz);

        return 0;
}
```

```
int xyz_probe(struct platform_device *pdev)
{
        struct device *dev = &pdev->dev;
        struct xyz_data *xyz;

        xyz = devm_kzalloc(dev,sizeof(*zyx), GFP_KERNEL);
        if (!xyz)
                return -ENOMEM;

        xyz->foo = devm_alloc_foo(dev);
        if (IS_ERR(xyz->foo))
                return PTR_ERR(xyz->foo);

        xyz->bar = devm_alloc_bar(dev);
        if (IS_ERR(xyz->bar))
                return PTR_ERR(xyz->bar);

        xyz->baz = devm_alloc_baz(dev);
        if (IS_ERR(xyz->baz))
                return PTR_ERR(xyz->baz);

        platform_set_drvdata(pdev, xyz);

        return 0;
}
```

Problem?



YouTube FR
Szukaj
Why is devm_kzalloc() harmful and what can we do about it
Laurent Pinchart
Linux Plumbers Conference | Dublin, Ireland Sept. 12-14, 2022
0:01 / 43:29
2
Udostępnij
Pobierz
Klip
Zapisz
Subskrybuj

Linux Plumbers Conference 2022

Why is devm_kzalloc() harmful and what can we do about it - Laurent Pinchart

Linux Plumbers Conference
6,47 tys. subskrybentów

https://www.youtube.com/watch?v=kW8LHWlJPTU

Problem?

From Laurent Pinchart <>
Subject Is devm_* broken ?
Date Wed, 15 Jul 2015 01:34:53 +0300

Hello,

I came to realize not too long ago that the following sequence of events will lead to a crash with any platform driver that uses devm_* and creates device nodes.

1. Get a platform device bound it its driver
2. Open the corresponding device node in userspace and keep it open
3. Unbind the platform device from its driver through sysfs

```
echo <device-name> > /sys/bus/platform/drivers/<driver-name>/unbind
```

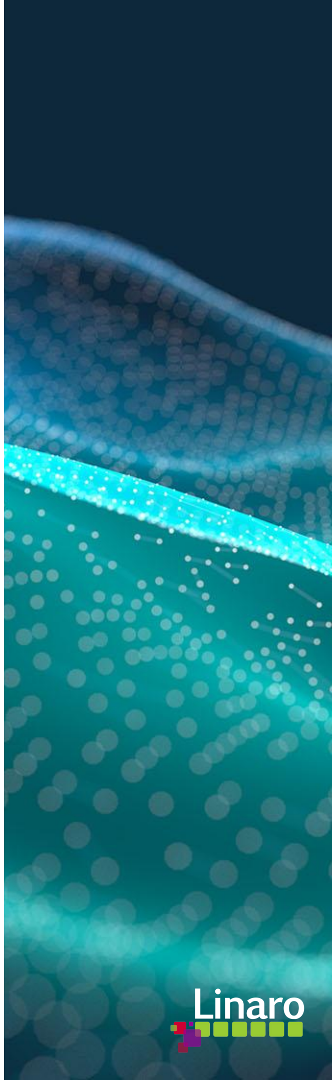
(or for hotpluggable devices just unplug the device)

4. Close the device node
5. Enjoy the fireworks

Problem?

Gist: drivers use `devm_kzalloc() to` allocate structures that should not be freed at driver unbind but instead live for as long as they are referenced

```
> How is this different from the free happening explicitly in the remove
> function?

It's not.  The real problem is that people don't understand life time
rules and expect magic interfaces to fix it for them.
```



Problem?

`expect magic interfaces to fix it`

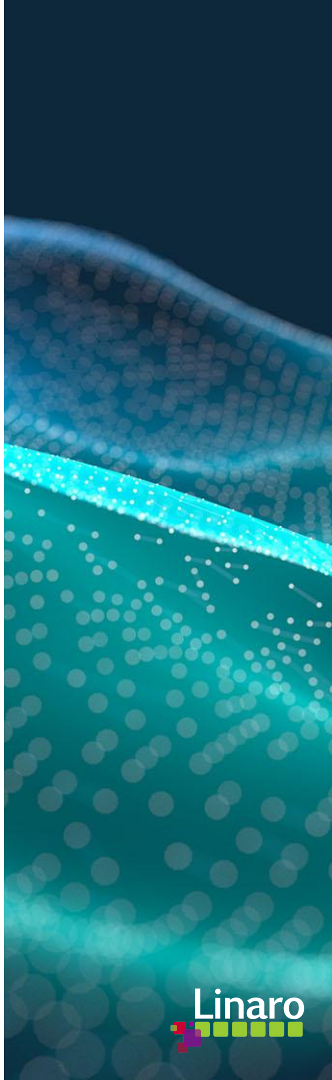
Problem?

- GPIO character device does indeed crash
- I2C character device... deadlocks

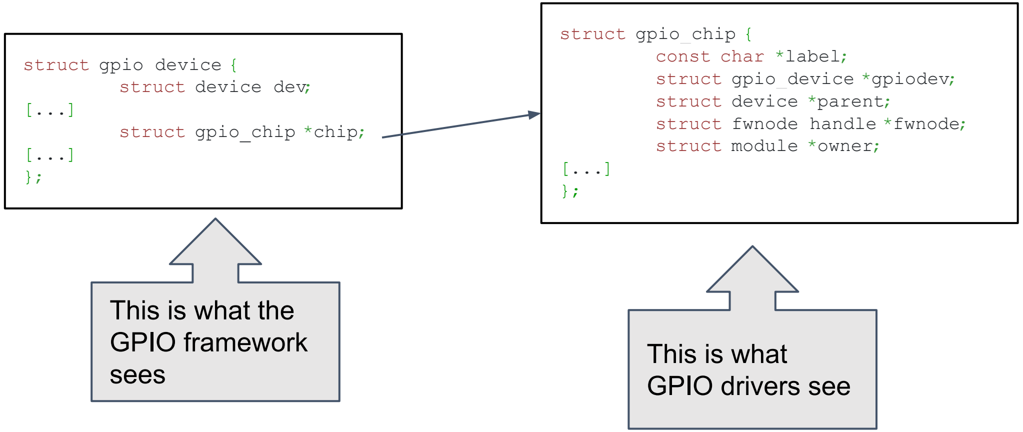
But...

- UART works just fine and gracefully returns an error to user-space

How come?



Let’s investigate GPIO!



```
struct gpio device {
        struct device dev;
[...]
        struct gpio_chip *chip;
[...]
};
```

```
struct gpio_chip {
        const char *label;
        struct gpio_device *gpiodev;
        struct device *parent;
        struct fwnode handle *fwnode;
        struct module *owner;
[...]
};
```

This is what the GPIO framework sees

This is what GPIO drivers see

Let’s investigate GPIO!

Crash happens in gpiolib.c at line:

```
2695                    struct gpio_chip *gc = desc_array[i]->gdev->chip;
```

Because:

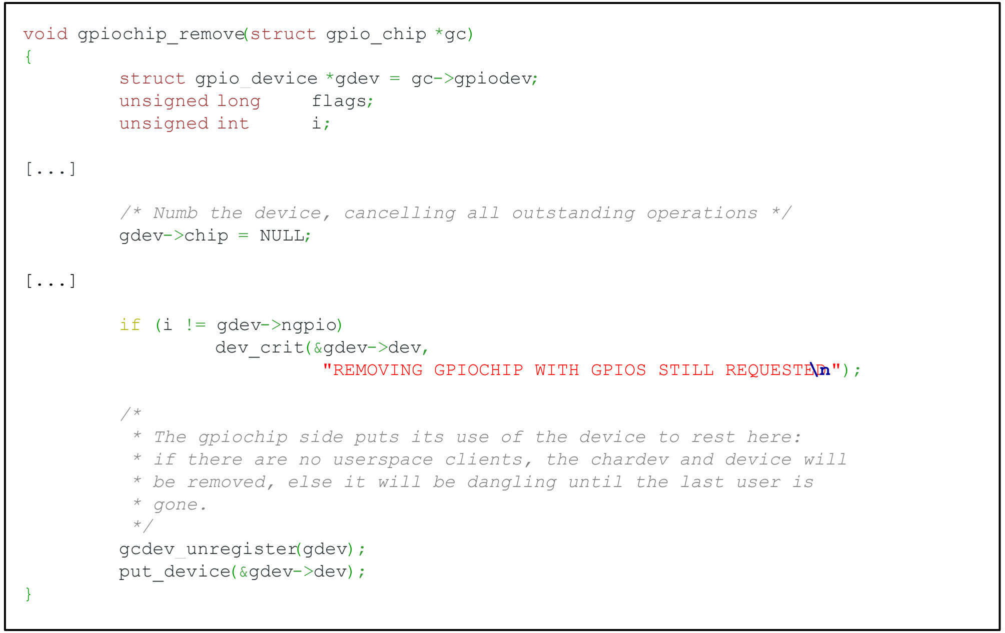
```
gdev->chip == NULL
```

We never check if gdev->chip == NULL!

```
static long linereq_ioctl(struct file *file, unsigned int cmd,
                            unsigned long arg)
{
        struct linereq *lr = file->private_data;
        void __user *ip = (void __user *)arg;

        switch (cmd) {
        case GPIO_V2_LINE_GET_VALUES_IOCTL:
                return linereq_get_values(lr, ip);
        case GPIO_V2_LINE_SET_VALUES_IOCTL:
                return linereq_set_values(lr, ip);
        case GPIO_V2_LINE_SET_CONFIG_IOCTL:
                return linereq_set_config(lr, ip);
        default:
                return -EINVAL;
        }
}
```

```
static const struct file_operations
gpio_fileops = {
        .release = gpio_chrdev_release,
        .open = gpio_chrdev_open,
        .poll = lineinfo_watch_poll,
        .read = lineinfo_watch_read,
        .owner = THIS_MODULE,
        .llseek = no_llseek,
        .unlocked_ioctl = gpio_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl = gpio_ioctl_compat,
#endif
};
```

Let’s investigate GPIO!



```
void gpiochip_remove(struct gpio_chip *gc)
{
        struct gpio_device *gdev = gc->gpiodev;
        unsigned long     flags;
        unsigned int      i;

[...]

        /* Numb the device, cancelling all outstanding operations */
        gdev->chip = NULL;

[...]

        if (i != gdev->ngpio)
                dev_crit(&gdev->dev,
                           "REMOVING GPIOCHIP WITH GPIOS STILL REQUESTED\n");

        /*
         * The gpiochip side puts its use of the device to rest here:
         * if there are no userspace clients, the chardev and device will
         * be removed, else it will be dangling until the last user is
         * gone.
         */
        gcdev_unregister(gdev);
        put_device(&gdev->dev);
}
```

Let’s investigate I2C!

```
void i2c_del_adapter(struct i2c_adapter *adap)
{
[...]

          /* wait until all references to the device are gone
           *
           * FIXME: This is old code and should ideally be replaced by an
           * alternative which results in decoupling the lifetime of the struct
           * device from the i2c_adapter, like spi or netdev do. Any solution
           * should be thoroughly tested with DEBUG_KOBJECT_RELEASE enabled!
           */
          init completion(&adap->dev released);
          device_unregister(&adap->dev);
          wait_for_completion(&adap->dev_released);

[...]
}
```

```
static void i2c_adapter_dev_release(struct device *dev)
{
          struct i2c_adapter *adap = to_i2c_adapter(dev);
          complete(&adap->dev_released);
}
```

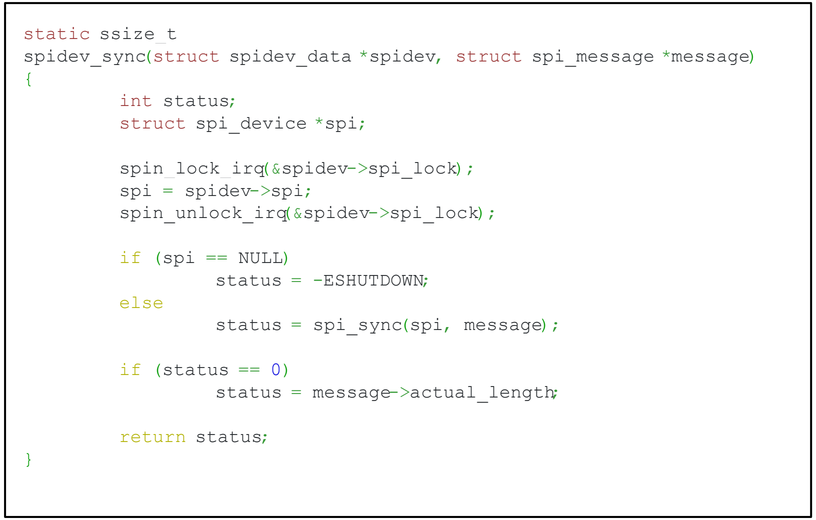
`i2c_adapter_dev_release()` is not called as long as there are open file descriptors and so `i2c_del_adapter()` waits waits forever

Why does UART work?

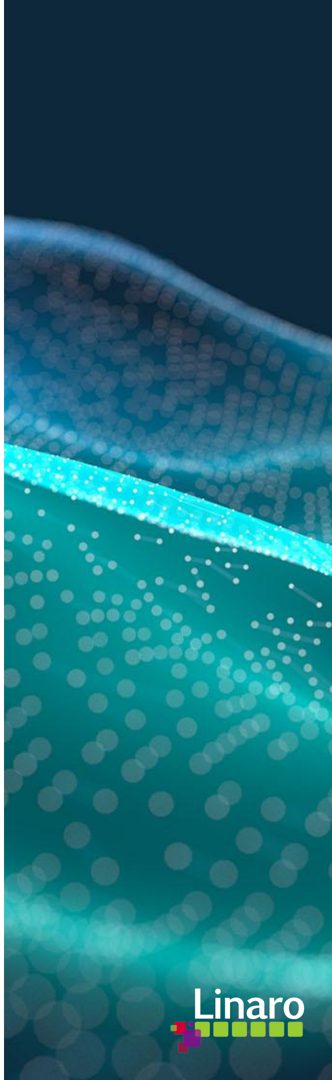
```
static int uart_write(struct tty_struct *tty, const unsigned char *buf, int count)
{
[...]

        port = uart_port_lock(state, flags);
        circ = &state->xmit;
        if (!circ->buf) {
                uart_port_unlock(port, flags);
                return 0;
        }

        while (port) {
[...]
        }

        __uart_start(tty);
        uart_port_unlock(port, flags);
        return ret;
}
```

```
#define uart_port_lock(state, flags)                                    \
        ({                                                              \
                struct uart_port *__uport = uart_port_ref(state);      \
                if (__uport)                                            \
                        spin_lock_irqsave(&__uport->lock, flags);      \
                __uport;                                                \
        })
```

Let’s investigate SPI!



```
static ssize_t
spidev_sync(struct spidev_data *spidev, struct spi_message *message)
{
        int status;
        struct spi_device *spi;

        spin_lock_irq(&spidev->spi_lock);
        spi = spidev->spi;
        spin_unlock_irq(&spidev->spi_lock);

        if (spi == NULL)
                status = -ESHUTDOWN;
        else
                status = spi_sync(spi, message);

        if (status == 0)
                status = message->actual_length;

        return status;
}
```



Let’s fix GPIO

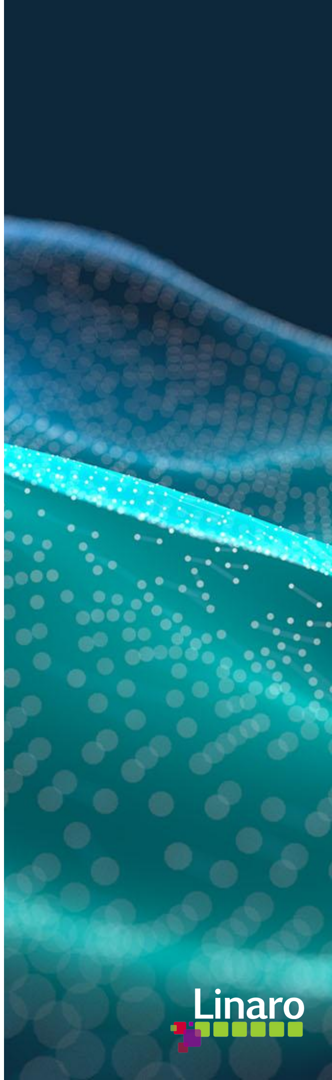
- GPIO
  - Check if gdev->chip != NULL and protect from concurrent access:
    - `533aae7c94db ("gpiolib: cdev: fix NULL-pointer dereferences")`
    - `bdbbae241a04 ("gpiolib: protect the GPIO device against being dropped while in use by user-space")`

Let’s fix SPI!

- SPI
  - Fix the race condition, replace spinlock with mutex and extend the critical sections:
    - `a720416d9463 ("spi: spidev: fix a race condition when accessing spidev->spi")`
    - `6b35b173dbc1 ("spi: spidev: remove debug messages that access spidev->spi without locking")`
    - `9bab63a3e949 ("spi: spidev: fix a recursive locking error")`

Let’s (try to) fix I2C!

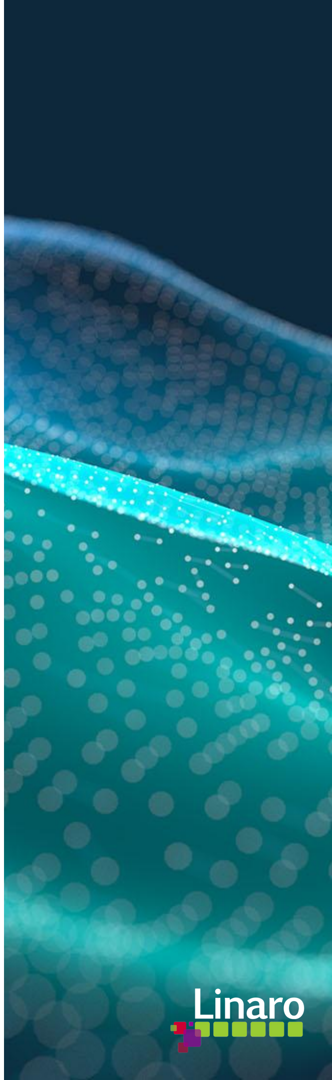
- I2C
  - Drop the completion and check if the adapter exists, protect from concurrent access:
    - `Commit ("i2c: dev: don't allow user-space to deadlock the kernel")`



Let’s (try to) fix I2C!

- ~~I2C~~
  - ~~Drop the completion and check if the adapter exists, protect from concurrent access:~~
    - ~~`Commit ("i2c: dev: don't allow user-space to deadlock the kernel")`~~

`struct i2c_adapter` (embedding i2c’s `struct device`) is allocated by bus drivers! It’s removed in `.remove()` and so we must not reference it after the driver unbinds!

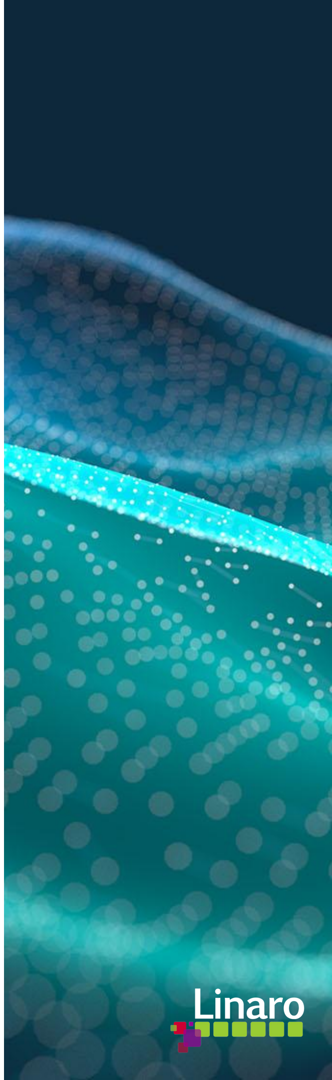


Don’t let drivers allocate and control the lifetime of `struct device` - leave it to subsystems

Some subsystems get it right

- GPIO is fine - struct gpio_device (embedding struct device) is allocated- and its lifetime managed by the subsystem
- UART and watchdog (and probably many others) are fine
- SPI drivers allocate struct device with `spi_alloc_master()` and then handle over its management to spi subsystem (?)
- Many more can be vulnerable!
- Some subsystems get the object lifetime right but still suffer from race conditions

It’s all about the logical scope of objects



DRM? Media?

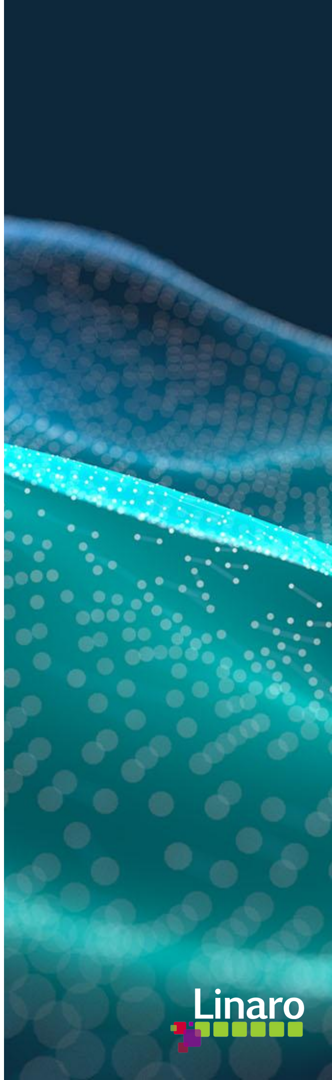
- Both suffer from race conditions in syscall handling
- Both also require drivers to manage `struct device`
- DRM is worse as `struct file_operations` is not centralized

So... is devres safe?

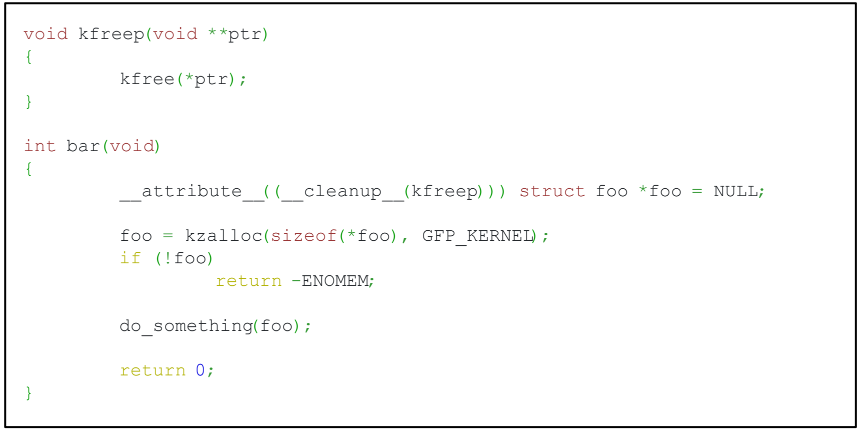
- No evidence that it isn’t
- If a resource can be released in driver’s `.remove()`, it can be managed by devres
- Need to pay attention to cross-subsystem interactions
- `devm_krealloc()` needs semantic clarification
- Devres makes code safer, more reliable and easier to read!
- Devres has a very limited scope

What are the alternatives/supplements?

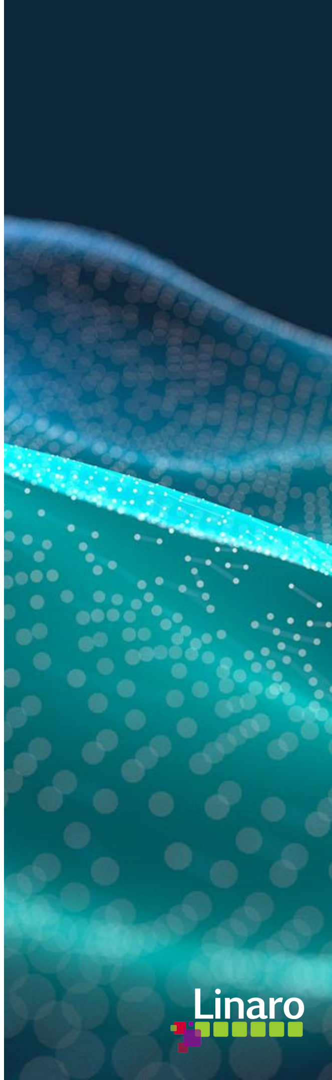
- Using Rust
- Introducing `__attribute__((__cleanup__(func)))` to the kernel
- Using the above in conjunction with reference counting



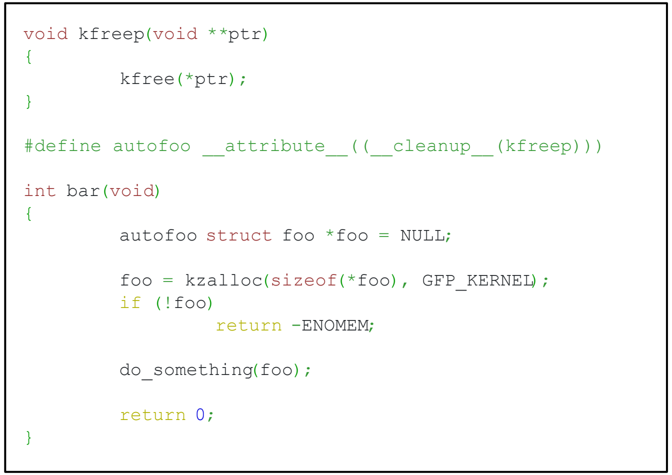
What are the alternatives?



```
void kfreep(void **ptr)
{
        kfree(*ptr);
}

int bar(void)
{
        __attribute__((__cleanup__(kfreep))) struct foo *foo = NULL;

        foo = kzalloc(sizeof(*foo), GFP_KERNEL);
        if (!foo)
                return -ENOMEM;

        do_something(foo);

        return 0;
}
```

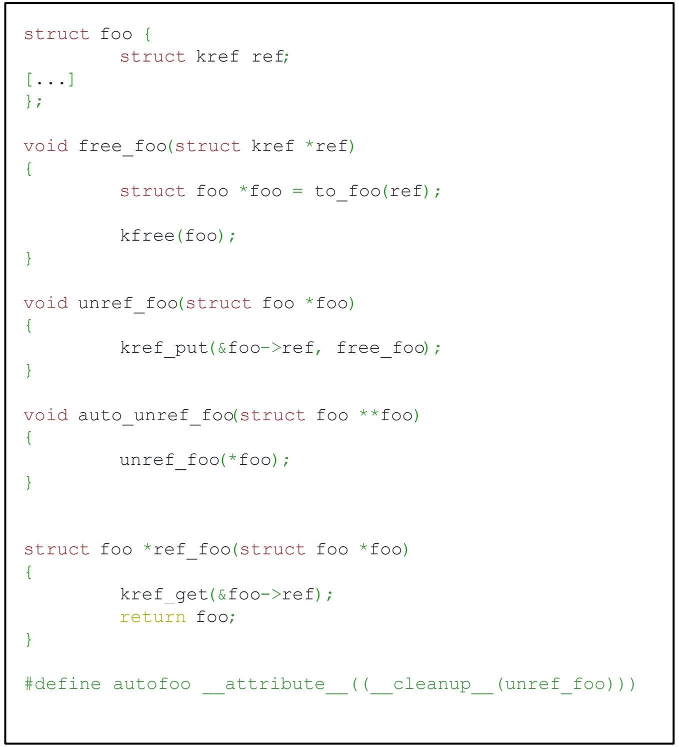


What are the alternatives?

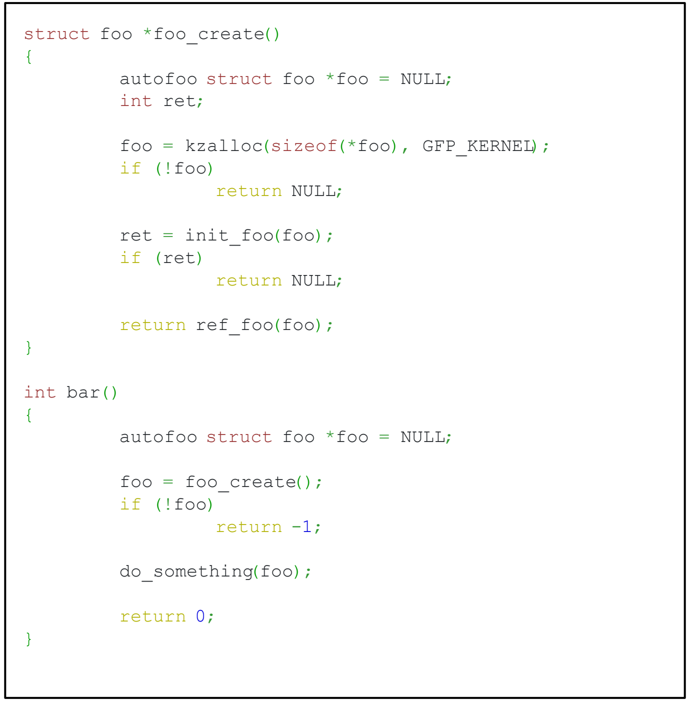


```
void kfreep(void **ptr)
{
        kfree(*ptr);
}

#define autofoo __attribute__((__cleanup__(kfreep)))

int bar(void)
{
        autofoo struct foo *foo = NULL;

        foo = kzalloc(sizeof(*foo), GFP_KERNEL);
        if (!foo)
                return -ENOMEM;

        do_something(foo);

        return 0;
}
```

What are the alternatives?



```
struct foo {
        struct kref ref;
[...]
};

void free_foo(struct kref *ref)
{
        struct foo *foo = to_foo(ref);

        kfree(foo);
}

void unref_foo(struct foo *foo)
{
        kref_put(&foo->ref, free_foo);
}

void auto_unref_foo(struct foo **foo)
{
        unref_foo(*foo);
}


struct foo *ref_foo(struct foo *foo)
{
        kref_get(&foo->ref);
        return foo;
}

#define autofoo __attribute__((__cleanup__(unref_foo)))
```



```
struct foo *foo_create()
{
        autofoo struct foo *foo = NULL;
        int ret;

        foo = kzalloc(sizeof(*foo), GFP_KERNEL);
        if (!foo)
                return NULL;

        ret = init_foo(foo);
        if (ret)
                return NULL;

        return ref_foo(foo);
}

int bar()
{
        autofoo struct foo *foo = NULL;

        foo = foo_create();
        if (!foo)
                return -1;

        do_something(foo);

        return 0;
}
```

Thank you

Q & A

Linaro