Linux Plumbers Conference | September 12-14, 2022
Why is devm_kzalloc() harmful and what can we do about it
Laurent Pinchart – Ideas on Board

Device resource management

commit 9ac7849e35f705830f7b016ff272b0ff1f7ff759 (v2.6.21)
Author: Tejun Heo <htejun@gmail.com>
Date: Sat Jan 20 16:00:26 2007 +0900

    devres: device resource management

    Implement device resource management, in short, devres. A device
    driver can allocate arbirary size of devres data which is associated
    with a release function. On driver detach, release function is
    invoked on the devres data, then, devres data is freed.

Device resource management

Initially came with a limited set of devm helpers:

- I/O remap (devm_ioport_map, devm_ioremap, devm_ioremap_nocache, pcim_iomap, pcim_iomap_table)
- IRQ (devm_request_irq)
- DMA memory (dmam_alloc_coherent, dmam_alloc_noncoherent, dmam_declare_coherent_memory, dmam_pool_create)
- SLAB allocation (**devm_kzalloc**)

IDEAS ON BOARD

First devm_kzalloc conversion

First conversion to **devm_kzalloc** in v2.6.28:

commit 3136e903fa2d493ebc1b8a8fbdde2d3a17f85acd
Author: Atsushi Nemoto <anemo@mba.ocn.ne.jp>
Date: Wed Nov 26 10:26:29 2008 +0000

    [MTD] physmap: fix memory leak on physmap_flash_remove by using devres

    physmap_flash_remove releases only last memory region. This causes
    memory leak if multiple resources were provided.

    This patch fixes this leakage by using devm_ functions.

IDEAS ON BOARD

Typical conversion to devm_*

```
diff --git a/drivers/media/i2c/mt9p031.c b/drivers/media/i2c/mt9p031.c
index e32833262d32..e0bad594c8da 100644
--- a/drivers/media/i2c/mt9p031.c
+++ b/drivers/media/i2c/mt9p031.c
@@ -927,7 +927,7 @@ static int mt9p031_probe(struct i2c_client *client,
                return -EIO;
        }

-       mt9p031 = kzalloc(sizeof(*mt9p031), GFP_KERNEL);
+       mt9p031 = devm_kzalloc(&client->dev, sizeof(*mt9p031), GFP_KERNEL);
        if (mt9p031 == NULL)
                return -ENOMEM;



@@ -1001,8 +1001,8 @@ static int mt9p031_probe(struct i2c_client *client,
        mt9p031->format.colorspace = V4L2_COLORSPACE_SRGB;

        if (pdata->reset != -1) {
-               ret = gpio_request_one(pdata->reset, GPIOF_OUT_INIT_LOW,
-                                      "mt9p031_rst");
+               ret = devm_gpio_request_one(&client->dev, pdata->reset,
+                                           GPIOF_OUT_INIT_LOW, "mt9p031_rst");
                if (ret < 0)
                        goto done;
```

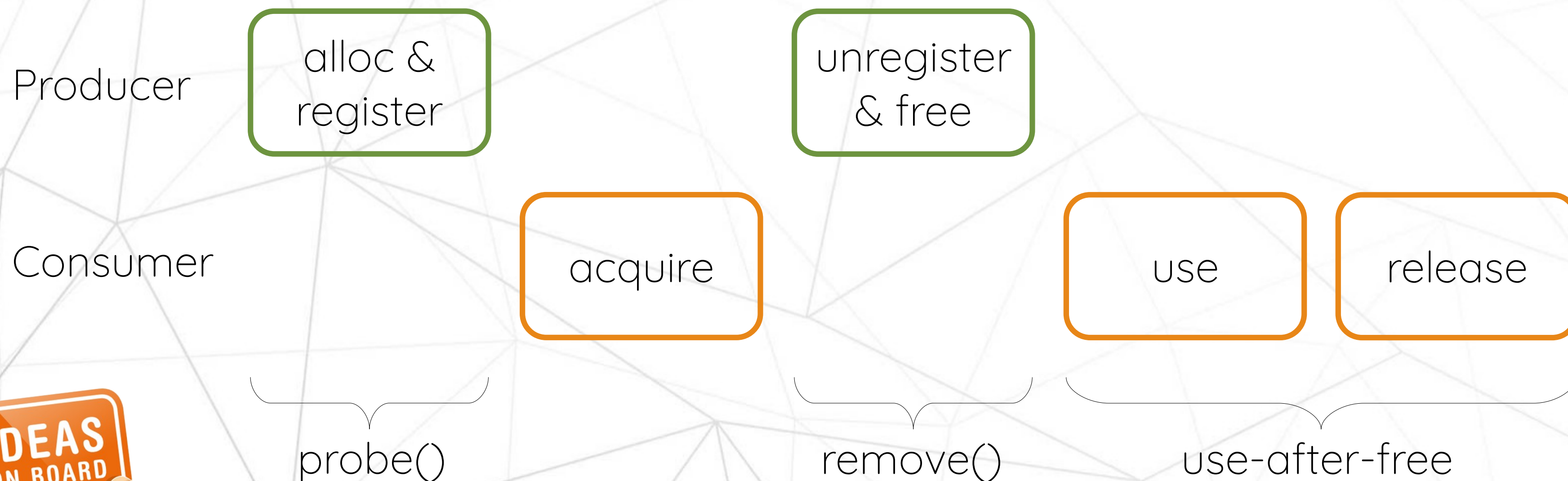
```
@@ -1013,12 +1013,8 @@ static int mt9p031_probe(struct i2c_client
*client,

 done:
        if (ret < 0) {
-               if (mt9p031->reset != -1)
-                       gpio_free(mt9p031->reset);
-
                v4l2_ctrl_handler_free(&mt9p031->ctrls);
                media_entity_cleanup(&mt9p031->subdev.entity);
-               kfree(mt9p031);
        }

        return ret;
@@ -1032,9 +1028,6 @@ static int mt9p031_remove(struct i2c_client
*client)
        v4l2_ctrl_handler_free(&mt9p031->ctrls);
        v4l2_device_unregister_subdev(subdev);
        media_entity_cleanup(&subdev->entity);
-       if (mt9p031->reset != -1)
-               gpio_free(mt9p031->reset);
-       kfree(mt9p031);

        return 0;
 }
```

A typical conversion to devm_* helpers doesn’t conceptually introduce bugs.

Linux Plumbers Conference 2022
>> Dublin, Ireland / September 12-14, 2022

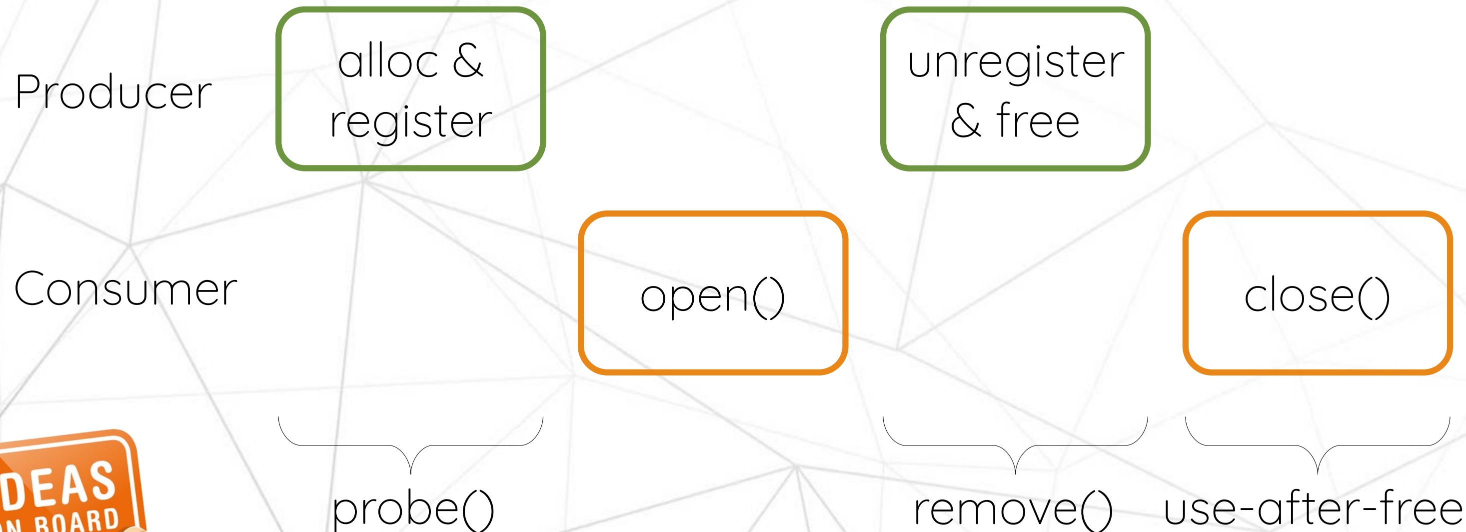
Use-after-free

A typical conversion to devm_* helpers doesn't conceptually **introduce** bugs, because **the bugs have been there all along**. If a resource is **freed** at remove time (a.k.a. detach, a.k.a. disconnect, a.k.a. unbind), a use-after-free may occur as consumers may hold references.



Use-after-free

In the particular (and particularly common) case of usage from userspace through a chardev, the acquire step is typically an open() and the release step a close(). Crashes can easily be triggered by userspace, and occur in the file_operation .release() handler in response to close().



Use-after-free

Resources must not be freed at remove time, but only after the last reference to the resource has been released.

References can be held in the kernel (e.g. clk, gpio, ...) or in userspace (e.g. opened fds).

devres release – The promise

commit 9ac7849e35f705830f7b016ff272b0ff1f7ff759
Author: Tejun Heo <htejun@gmail.com>
Date: Sat Jan 20 16:00:26 2007 +0900

devres: device resource management

Implement device resource management, in short, devres. A device driver can allocate arbirary size of devres data which is associated with a release function. **On driver detach**, release function is invoked on the devres data, then, devres data is freed.

IDEAS ON BOARD

devres release – What was heard

commit 9ac7849e35f705830f7b016ff272b0ff1f7ff759
Author: Tejun Heo <htejun@gmail.com>
Date: Sat Jan 20 16:00:26 2007 +0900

devres: device resource management

Implement device resource management, in short, devres. A device driver can allocate arbirary size of devres data which is associated with a release function. **By magic, at the right time**, the release function is invoked on the devres data, then, devres data is freed.

devm_* takes over the kernel

Over time, has grown to ~170 (as of v6.0-rc5) devres_alloc* calls, split in four big categories:

- Resource allocation (devm_input_allocate_device, devm_phy_create, devm_kzalloc, …)
- Resource registration (e.g. devm_clk_register, devm_watchdog_register_device, …)
- Resource acquisition (e.g. devm_ioremap, devm_clk_get, devm_gpio_request, …)
- (Ab)use of devres for generic association of data with a struct device (firmware_request_cache, component framework, …)

A shortage of magic

4 PRIVET DRIVE
NO MAGIC ALLOWED

IDEAS ON BOARD

Not all usage of devm_* is bad

Acquiring resources that must not be touched after unbind can use devm_* safely[1] as a use-after-free would then be a bug in the driver. For instance, devm_ioremap_resource() is fine, as the driver model prohibits drivers from touching memory-mapped I/O after unbind.

Safety of other usages range from simply scary to crossing the Niagara falls on a tightrope with shoelaces tied together. Handling resource **registration** with devm_* leads to unregistration after the .remove() handler returns, while in many case drivers should unregister the resource to avoid new users before handling other cleanups.

IDEAS ON BOARD

[1] Actually, mostly safely, as resource release ordering needs to be carefully considered. There's a risk of, for instance, shutting down the PM domain before IRQs are freed.

Problems – Solutions

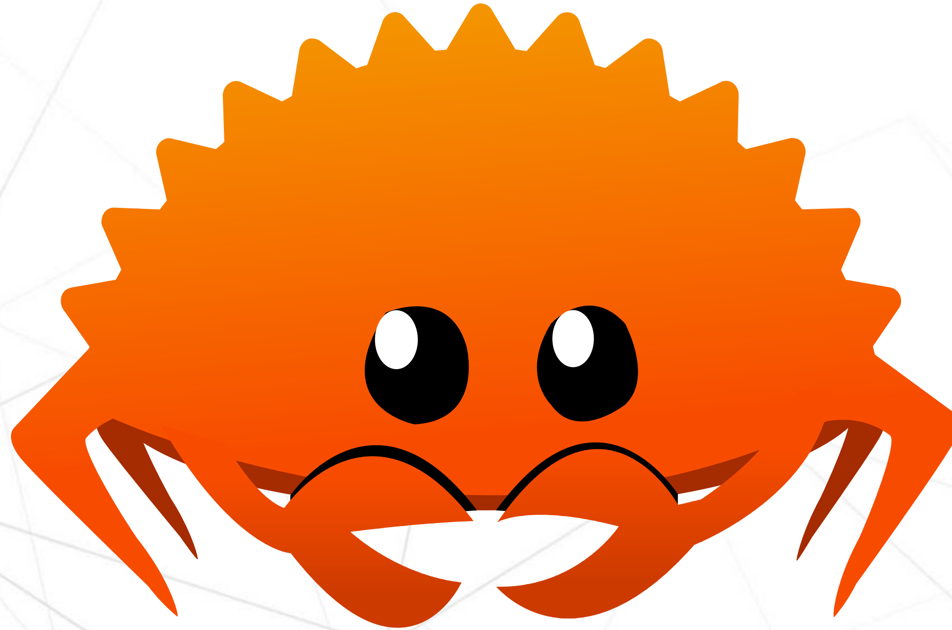
We have carefully ignored those issue for a long time (cfr a discussion of the same topic on the ksummit-discuss mailing list in 2015[1]). Lifetime management problems now plague many areas of the kernel, with no hope of solving all problems in one go.

Let's focus on the **unbind/close race** involving userspace. It is especially dangerous as it can be **triggered by unprivileged userspace** with hot-pluggable devices, and could be a (relatively) low-hanging fruit compared to the other problems. How do we fix that particular use-after-free problem ?

IDEAS ON BOARD

[1] https://lore.kernel.org/all/2111196.TG1k3f53YQ@avalon/

Rewrite the whole kernel in Rust?



IDEAS ON BOARD

Can't realistically be considered a "low-hanging fruit".

Never free resources?

An obvious solution to the use-after-free problem is to remove the “free” (assuming we can’t remove the “use”). This may be an option for low-level resources such as clocks or GPIOs, especially given that the corresponding in-kernel APIs have no way to notify the consumer of resource removal.

This can’t be a universal solution, as it would effectively leak memory on module removal (not great), device unbind (bad) or device unplug (abysmal).

Reference counting?

If we can’t drop the “free”, the only other option is to delay until after the “use”. This involves reference-counting the resource. Multiple options exist today in the kernel.

IDEAS ON BOARD

Manual reference counting?

```
/**
 * struct device - The basic device structure
 * [...]
 * @release:    Callback to free the device after all references have
 *              gone away. This should be set by the allocator of the
 *              device (i.e. the bus driver that discovered the device).
 * [...]
 */
struct device {
        [...]
        void    (*release)(struct device *dev);
        [..]
};

struct video_device {
        [...]
        struct device dev;
        [...]
        void (*release)(struct video_device *vdev);
        [...]
};

/* struct video_device .release() handler */
static void uvc_release(struct video_device *vdev)
{
        struct uvc_streaming *stream = video_get_drvdata(vdev);
        struct uvc_device *dev = stream->dev;

        kref_put(&dev->ref, uvc_delete);
}
```

The mechanism exists in struct device already. It need to be propagated across the structure inheritance chain all the way to individual drivers, pushing the complexity to the leaf nodes.

IDEAS ON BOARD

Linux Plumbers Conference 2022

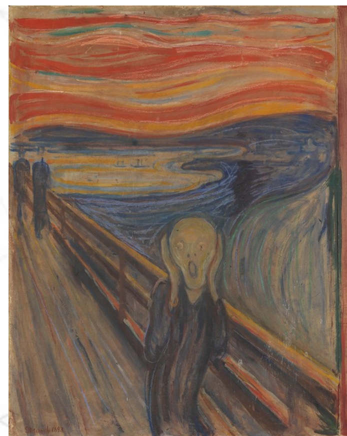
>> Dublin, Ireland / September 12-14, 2022

Manual reference counting?

```
/**
 * video_device_release - helper function to release &struct video_device
 *
 * @vdev: pointer to &struct video_device
 *
 * Can also be used for video_device->release\(\).
 */
void video_device_release(struct video_device *vdev);

/**
 * video_device_release_empty - helper function to implement the
 *      video_device->release\(\) callback.
 *
 * @vdev: pointer to &struct video_device
 *
 * This release function does nothing.
 *
 * It should be used when the video_device is a static global struct.
 *
 * .. note::
 *      Having a static video_device is a dubious construction at best.
 */
void video_device_release_empty(struct video_device *vdev);
```

Even subsystem cores get it wrong by providing bad helpers.

Can we get drivers to implement reference counting and release correctly, or is it a lost cause?



IDEAS ON BOARD

devres-like lifetime management?

```
/**
 * drmm_kzalloc - &drm_device managed kzalloc()
 * @dev: DRM device
 * @size: size of the memory allocation
 * @gfp: GFP allocation flags
 *
 * This is a &drm_device managed version of kzalloc(). The allocated memory is
 * automatically freed on the final drm_dev_put(). Memory can also be freed
 * before the final drm_dev_put() by calling drmm_kfree().
 */
static inline void *drmm_kzalloc(struct drm_device *dev, size_t size,
                                 gfp_t gfp)
{
        return drmm_kmalloc(dev, size, gfp | __GFP_ZERO);
}

/**
 * DOC: managed resources
 *
 * Inspired by struct &device managed resources, but tied to the lifetime of
 * struct &drm_device, which can outlive the underlying physical device,
 * usually when userspace has some open files and other handles to resources
 * still open.
 */
```

The DRM subsystem has managed helpers that mimic devm_* but tie the lifetime to the resource exposed to userspace (i.e. struct drm_device).

Problems:

- Code duplication (currently separate implementation devres, could possibly share code).
- Lifetime tied to one given resource, doesn't work for drivers that create multiple resources.

IDEAS ON BOARD

Garbage collection?

```
/**
 * struct device - The basic device structure
 * [...]
 * @release:    Callback to free the device after all references have
 *              gone away. This should be set by the allocator of the
 *              device (i.e. the bus driver that discovered the device).
 * [...]
 */
struct device {
        [...]
        void    (*release)(struct device *dev);
        [..]
};

struct video_device {
        [...]
        struct device dev;
        [...]
        void (*release)(struct video_device *vdev);
        [...]
};

/* struct video_device .release() handler */
static void uvc_release(struct video_device *vdev)
{
        struct uvc_streaming *stream = video_get_drvdata(vdev);
        struct uvc_device *dev = stream->dev;

        kref_put(&dev->ref, uvc_delete);
}
```

Could we simplify manual refcount handling with a new object that would

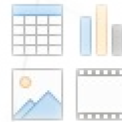
- store the kref release function pointer at initialization time
- store the parent-child relationships between objects
- automatically decrease reference counts based on those relationships

or would this be too close to a garbage collector to be acceptable?

IDEAS ON BOARD

Something else?

- Insert clever idea here



IDEAS ON BOARD

There's more – What, really?

This was the low-hanging fruit. Other problems that still need to be tackled are

- Ordering of the devres release, especially when mixing managed and non-managed resources.
- Removal of core resources (clocks, regulators, GPIOs, ...) with active users.
- Unbind/ongoing system call race (see [1] and [2]).

The last problem is a good candidate, as patches have been proposed and a consensus seems to have been reached. Discussion "just" died out.

[1] https://lore.kernel.org/linux-media/20171116003349.19235-2-laurent.pinchart+renesas@ideasonboard.com/

[2] https://lore.kernel.org/all/161117153248.2853729.2452425259045172318.stgit@dwillia2-desk3.amr.corp.intel.com/

IDEAS ON BOARD