6. Memory allocation

NXKR_YoungSikYang

Exported on 03/07/2024

Table of Contents

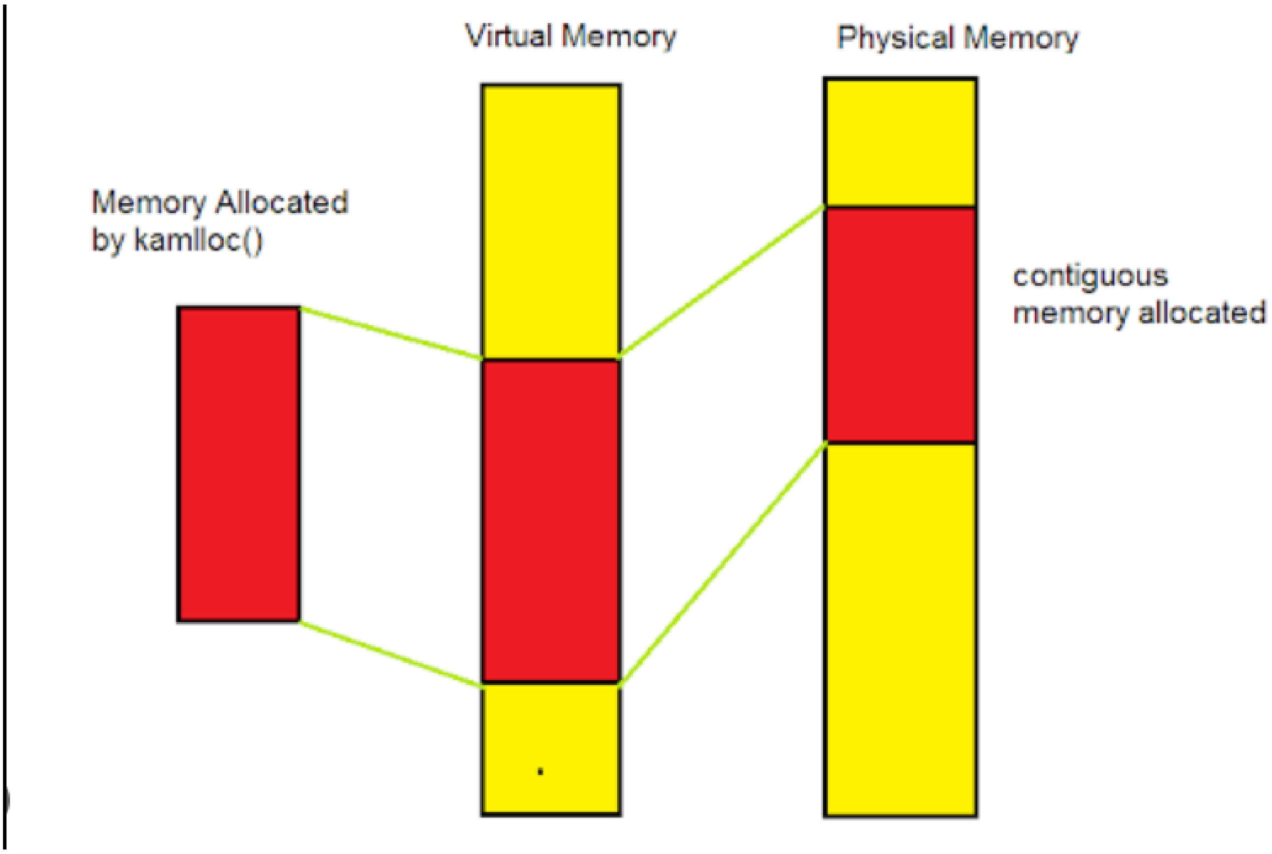
1. Memory allocation APIs

1.1 kmalloc()

- **Usage**: **kmalloc()** is used to allocate small to medium amounts of memory that are physically contiguous.
- **Allocation**: `void *kmalloc(size_t size, gfp_t flags);`
    - `size` is the amount of memory you want to allocate.
    - `flags` are the GFP (Get Free Pages) flags that affect the behavior of the allocation, like blocking or non-blocking.
- **Free**: `kfree(const void *ptr)` frees allocated memory

1.1.1 Memory allocated by kmalloc()

The memory allocated by kmalloc() is contiguous in the physical memory as well as in the virtual memory.



1.1.2 GFP flags

The flags used in kmalloc() are defined in `slap.h`

```
/**
 * kmalloc - allocate memory
 * @size: how many bytes of memory are required.
 * @flags: the type of memory to allocate.
 *
 * kmalloc is the normal method of allocating memory
 * for objects smaller than page size in the kernel.
 *
 * The @flags argument may be one of:
 *
 * %GFP_USER - Allocate memory on behalf of user.  May sleep.
 *
 * %GFP_KERNEL - Allocate normal kernel ram.  May sleep.
 *
 * %GFP_ATOMIC - Allocation will not sleep.  May use emergency pools.
 *   For example, use this inside interrupt handlers.
 *
 * %GFP_HIGHUSER - Allocate pages from high memory.
 *
 * %GFP_NOIO - Do not do any I/O at all while trying to get memory.
 *
 * %GFP_NOFS - Do not make any fs calls while trying to get memory.
 *
 * %GFP_NOWAIT - Allocation will not sleep.
 *
 * %__GFP_THISNODE - Allocate node-local memory only.
 *
 * %GFP_DMA - Allocation suitable for DMA.
 *   Should only be used for kmalloc() caches. Otherwise, use a
 *   slab created with SLAB_DMA.
 *
 * Also it is possible to set different flags by OR'ing
 * in one or more of the following additional @flags:
 *
 * %__GFP_COLD - Request cache-cold pages instead of
 *   trying to return cache-warm pages.
 *
 * %__GFP_HIGH - This allocation has high priority and may use emergency pools.
 *
 * %__GFP_NOFAIL - Indicate that this allocation is in no way allowed to fail
 *   (think twice before using).
 *
 * %__GFP_NORETRY - If memory is not immediately available,
 *   then give up at once.
 *
 * %__GFP_NOWARN - If allocation fails, don't issue any warnings.
 *
 * %__GFP_REPEAT - If allocation fails initially, try once more before failing.
 *
 * There are other flags available as well, but these are not intended
 * for general use, and so are not documented here. For a full list of
 * potential flags, always refer to linux/gfp.h.
 */
```

1.2 vmalloc()

- **Usage**: **vmalloc()** is used for allocating large amounts of memory. The memory is virtually contiguous but may not be physically contiguous, which is useful when large buffers are needed, and physical continuity is not a requirement.
- **Allocation**: `void *vmalloc(unsigned long size);`
- **Free**: `vfree(const void *ptr)` releases allocated memory.

1.3 dma-alloc()

- **Usage**: Used to allocate physically contiguous memory that is also DMA (Direct Memory Access) capable. There are devices that require DMA operations requiring contiguous physical memory.
- **Allocation**: `void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flags)`
    - `dev` is a device structure.
    - `size` is the allocation size.
    - `dma_handle` is a pointer to a DMA address(physical address).
- **Free**: `void dma_free_coherent(struct device *dev, size_t size, void *cpu_addr, dma_addr_t dma_handle)`
- **Cacheable/Non-Cacheable**: `dma_alloc_coherent` typically returns non-cacheable memory to ensure data consistency, as DMA operations directly access the memory, skipping the CPU and cache.

1.4 Other APIs

There are several other memory allocation methods used in the Linux kernel for specific purposes.

- get_free_pages()
- alloc_page()
- vmalloc_to_pfn()
- High Memory Access: kmap()

2. SLUB

The SLUB is a memory allocator used in the Linux kernel, designed to be simple and efficient. It was introduced to improve upon the shortcomings of previous allocators like SLAB and SLOB, particularly on multicore systems.

Features and Advantages

- **Efficiency**: SLUB minimizes the use of locks and reduces fragmentation, making it more efficient, especially in SMP (Symmetric Multiprocessing) environments.
- **Scalability**: It scales well with the number of CPUs, making it suitable for high-performance and high-throughput systems.
- **Simplicity**: Its design is simpler than that of SLAB, making it easier to maintain and understand.
- **Debugging Support**: SLUB comes with extensive debugging support to detect common errors such as double frees, memory overruns, and usage of freed objects.

How It Works

- **Caches and Objects**: SLUB operates by managing caches of objects, where each cache is tailored to a specific size of the object. This helps in optimizing memory usage and access speed.
- **Allocation**: When a request for memory allocation is made, SLUB tries to satisfy the request from the corresponding object cache. If there is a free object available, it is returned to the caller.
- **Freeing Memory**: When memory is freed, the object is returned to its cache, making it available for future allocations.
- **Per-CPU Caches**: To improve performance on multicore systems, SLUB maintains per-CPU caches, which reduce the need for locking and increase the speed of allocation and deallocation operations.

3. Practice

Code explanation

- Memory allocation and deallocation are executed on each button press by turns.
- **platform_driver** is used to map the DMA allocation to a (dummy)device that requires DMA operations.
- Each memory allocation API returns **NULL** if it failed.

bottom_half() is implemented in each practice section.

```
#include <linux/module.h>
#include <linux/platform_device.h>

#include <linux/gpio.h>
#include <linux/interrupt.h>

#include <linux/delay.h>
#include <linux/slab.h> // For kmalloc() and kfree()
#include <linux/vmalloc.h> // For vmalloc() and vfree()
#include <linux/dma-mapping.h> // For dma_alloc_coherent() and dma_free_coherent()
#include <linux/delay.h> // For mdelay()

#define button 3 // Example GPIO number
#define button2 36
#define LED1 4
#define LED2 5
#define ON 0
#define OFF 1

static struct device *dev;
static unsigned int irq_number;
static int toggle = 0;
static void *buffer;

// Interrupt handler function(top-half)
static irqreturn_t btn_irq_handler(int irq, void *dev_id)
{
    return IRQ_WAKE_THREAD;
}

static irqreturn_t bottom_half(int irq, void *dev_id);

static int set_irq(int gpio, irq_handler_t bottom_half, unsigned long flag)
{
    int result_irq;
    // Get the IRQ number for our GPIO
    irq_number = gpio_to_irq(gpio);
    if (irq_number < 0) {
```

```
        printk(KERN_INFO "GPIO to IRQ mapping failed\n");
        return 1;
    }
    // Request the IRQ line
    result_irq = request_threaded_irq(irq_number, btn_irq_handler, bottom_half, flag,
"btn_irq", NULL);
    if (result_irq) {
        printk(KERN_INFO "IRQ request failed\n");
        return 1;
    }
    return 0;
}

static int my_platform_probe(struct platform_device *pdev)
{
    pr_info("my_platform_probe\n");
    dev = &pdev->dev;
    return 0;
}

static int my_platform_remove(struct platform_device *pdev)
{
    dev_info(&pdev->dev, "Platform device removed\n");
    return 0;
}

static struct of_device_id testdev_of_match[] = {
    { .compatible = "yang,mydevice", },
    {},
};
MODULE_DEVICE_TABLE(of, testdev_of_match);

static struct platform_driver my_platform_driver = {
    .probe = my_platform_probe,
    .remove = my_platform_remove,
    .driver = {
        .name = "my_platform_driver",
        .of_match_table = testdev_of_match,
        .owner = THIS_MODULE,
    },
};

// Module initialization
static int __init testdev_init(void)
{
    platform_driver_register(&my_platform_driver);
    // Request GPIO
    int result_btn;
    result_btn = gpio_request(button, "sysfs");
    if (result_btn) {
        pr_info("Cannot request the LED GPIO\n");
        return 1;
    }
```

```
    set_irq(button, bottom_half, IRQF_TRIGGER_FALLING);
    return 0;
}

// Module exit
static void __exit testdev_exit(void)
{
    platform_driver_unregister(&my_platform_driver);
    free_irq(irq_number, NULL);
    gpio_free(button);
    gpio_free(LED1);
    gpio_free(LED2);
}

module_init(testdev_init);
module_exit(testdev_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Your Name");
MODULE_DESCRIPTION("Example platform device driver with DMA allocation");
```

3.1 kmalloc()

3.1.1 Source code

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    if (!toggle) {
        // Allocate memory using kmalloc
        buffer = kmalloc(sizeof(char), GFP_ATOMIC);
        if (!buffer)
            return -ENOMEM;
        printk(KERN_INFO "kmalloc allocated, Address: %p\n", (char *)buffer);
    } else {
        // Free memory
        kfree(buffer);
        printk(KERN_INFO "kmalloc freed, Address: %p\n");
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}
```

3.1.2 Result

```
root@s5p6818:~# [ 4128.076000] kmalloc allocated, Address: ffffffc00a3a2700
[ 4128.272000] kmalloc freed, Address: 0000000000000001
```

4.2 vmalloc()

4.2.1 Source code

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    if (!toggle) {
        // Allocate memory using vmalloc
        buffer = vmalloc(sizeof(char));
        *buffer = 0;
        if (!buffer)
            return -ENOMEM; // Return error if allocation fails
        printk(KERN_INFO "vmalloc allocated, Address: %p\n", buffer);
    } else {
        // Free memory
        vfree(buffer);
        printk(KERN_INFO "vmalloc freed, Address: %p\n", buffer);
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}
```

4.2.2 Result

```
cat /proc/vmallocinfo | grep bottom_half
```

```
root@s5p6818:~# [ 4248.104000] vmalloc allocated, Address: ffffff80090b6000
root@s5p6818:~# cat /proc/vmallocinfo | grep bottom_half
0xffffff80090b6000-0xffffff80090b8000    8192 bottom_half+0x20/0x88 [interrupt] pages=1 vmalloc
```

4.3 dma_alloc_coherent()

4.3.1 Source code

```
static irqreturn_t bottom_half(int irq, void *dev_id)
{
    dma_addr_t dma_handle;
    size_t size = 1024;
    // Allocate memory using dma_alloc_coherent
    if (!toggle) {
        buffer = dma_alloc_coherent(dev, size, &dma_handle, GFP_ATOMIC);
        if (!buffer) {
            printk(KERN_INFO "dma_alloc failed\n");
            return -ENOMEM; // Return error if allocation fails
        }
        printk(KERN_INFO "dma_alloc allocated, Address: %p", dma_handle);
    } else {
        // Free memory
        dma_free_coherent(dev, size, buffer, dma_handle);
        printk(KERN_INFO "dma_alloc freed, Address: %p", dma_handle);
    }
    toggle = !toggle;
    return IRQ_HANDLED;
}
```

4.3.2 Result

```
root@s5p6818:~# [ 4417.744000] dma_alloc allocated, Address: 000000004c800000
[ 4417.984000] dma_alloc freed, Address: ffffff80080e127c
```